A Finite Element Based Optimisation Tool for Electrical Machines

by

Stiaan Gerber

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in the Faculty of Engineering at Stellenbosch University

Supervisor: Mr. J.M. Strauss
Co-supervisor: Mr. P.J. Randewijk
Faculty of Engineering
Department of Electrical and Electronic Engineering

March 2011

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2011

Abstract

Knowledge of the magnetic fields in the domain of electrical machines is required in order to model machines accurately. It is difficult to solve these fields analytically because of the complex geometries of electrical machines and the non-linear characteristics of the materials used to build them. Thus, finite element analysis, which can be used to solve the magnetic field accurately, plays an important part in the design of electrical machines.

When designing electrical machines, the task of finding an optimal design is not simple because the performance of the machine has a non-linear dependence on many variables. In these circumstances, numerical optimisation using finite element analysis is the most powerful method of finding optimal designs.

In this thesis, the work of improving an existing finite element simulation package, formerly known as the *Cambridge* package among its users, and the use of this package in the optimisation of electrical machine designs, is presented. The work involved restructuring the original package, expanding its capabilities and coupling it to numerical optimisers. The developed finite element package has been dubbed *SEMFEM*: the Stellenbosch Electrical Machines Finite Element Method.

The *Cambridge* package employed the air-gap element method, first proposed by Razek et. al. [2], to solve the magnetic field for different positions of the moving component in a time-stepped finite element simulation. Because many new machine topologies have more than one air-gap, the ability to model machines with multiple air-gaps is important. The *Cambridge* package was not capable of this, but during the course of this work, the ability to model machines with multiple air-gaps using the air-gap element method was implemented.

Many linear electrical machines have tubular, axisymmetric topologies. The functionality to simulate these machines was newly implemented because the original program was not capable of analysing these machines. Amongst other things, this involved the derivation of the coefficients of an axisymmetric air-gap element's stiffness matrix. This derivation, along with the original air-gap element derived by Razek et. al. [2] and the extension of the method to the Cartesian coordinate system by Wang et. al. [29, 30], completes the derivation of all two-dimensional air-gap elements.

In order to speed the numerical optimisation process, which is computationally expensive, parallelisation was introduced in two areas: at the level of the finite element simulation and at the level of the optimisation program.

The final product is a more powerful, more usable package, geared for the optimisation of electrical machines.

Uittreksel

Kennis van die magnetiese velde in die gebied van elektriese masjiene word benodig om masjiene akkuraat te modelleer. Dit is moeilik om hierdie velde analities op te los as gevolg die komplekse geometrieë van elektriese masjiene en die nie-lineêre karakteristieke van die materiale wat gebruik word om hulle te bou. Dus speel eindige element analise 'n belangrike rol in die ontwerp van elektriese masjiene omdat dit gebruik kan word om die magnetiese veld akkuraat te bepaal.

Wanneer elektriese masjiene ontwerp word, is dit nie 'n eenvoudige taak om 'n optimale ontwerp te vind nie omdat die werkverrigting van die masjien nie-lineêr afhanklik is van baie veranderlikes. Onder hierdie omstandighede is numeriese optimering, tesame met eindige element analise, die kragtigste metode om optimale ontwerpe te vind.

In hierdie tesis word die verbetering van 'n bestaande eindige element simulasie pakket, wat onder gebruikers van die pakket as die *Cambridge* pakket bekend staan, en die gebruik van hierdie pakket vir die optimering van elektriese masjiene, voorgelê. Die werk het die herstrukturering van die oorspronklike pakket, die uitbreiding van die pakket se vermoëns en die koppeling van die pakket aan numeriese optimeerders behels. Die ontwikkelde eindige element pakket word *SEMFEM* genoem: die Stellenbosch Elektriese Masjiene *Finite Element Method*.

Die *Cambridge* pakket het van die lugspleet element metode, soos oorspronlik deur Razek et. al. [2] voorgestel, gebruik gemaak om die magnetiese veld vir verskillende posisies van die bewegende komponent in 'n tyd-stapsgewyse eindige element simulasie op te los. Omdat baie nuwe masjien topologieë meer as een lugspleet het, is die vermoë om masjiene met meer as een lugspleet te kan modelleer belangrik. Die *Cambridge* pakket was nie hier toe in staat nie, maar die vermoë om masjiene met meervoudige lugsplete te modelleer is gedurende hierdie werk geïmplementeer.

Baie lineêre masjiene het tubulêre, assimmetriese topologieë. Die funksionaliteit om hierdie masjiene te simuleer is nuut geïmplementeer omdat die oorspronlike program nie in staat was om hierdie masjiene te analiseer nie. Dit het onder andere behels dat die koeffisiënte van 'n assimmetriese lugspleetelement se styfheidsmatriks afgelei moes word. Hierdie afleiding, tesame met die oorspronlike lugspleetelement afgelei deur Razek et. al. [2]

en die uitbreiding na die Cartesiese koördinaatstelsel deur Wang et. al. [29, 30], voltooi die afleiding van alle twee-dimensionele lugspleet elemente.

Om die numeriese optimeringsproses, wat tipies tydsgewys duur is, te versnel, is parallellisering op twee vlakke ingebring: op die vlak van die eindige element simulasie en op die vlak van die optimeringsprogram.

Die finale produk is 'n kragtiger, meer bruikbare pakket, goed aangepas vir die optimering van elektriese masjiene.

Acknowledgements

I would very much like to thank the following people and institutions for their support:

Mr. Johann Strauss, my supervisor, for his guidance. I think very few students are privileged to see their supervisors as often as I have.

Dr. Roger Wang, for his expertise and support.

Mr. Peter-Jan Randewijk, for introducing me to the worlds of Python, Linux, LaTeX and M4 circuit macros and his willingness to help.

The Centre for Renewable and Sustainable Energy Studies, for valuable support, financially and otherwise.

I once heard someone say that the acknowledgements page is where you thank your mother, your father and so on and traditionally, God. I would very much like to acknowledge my God, but I do not do this out of tradition. I really want to thank Him, for the life He has given me, for His grace and His love. And although the purpose of this thesis is partly to show how clever I am, I acknowledge that in truth, I am not really that clever and that I need Jesus.

Dedications

```
Vir my ma, wat my leer programmeer het;
// For my mother, who taught me to program
```

Contents

Nomenclature

Electromagnetics

$\mathbf{A}$	Magnetic vector potential
$\mathbf{J}$	Current density
μ	Permeability
κ	Reluctivity
$\mathbf{H}$	Magnetic field intensity
$\mathbf{B}$	Magnetic flux density
λ	Flux linkage
$\mathbf{M}_0$	Remanent flux density

The finite element method

$\mathbf{K}$	Global stiffness matrix
$\mathbf{K}^e$	Local stiffness matrix
$\mathbf{u}$	Global vector of (unknown) nodal vector potentials
$\mathbf{u}^e$	Local vector of (unknown) nodal vector potentials
$\mathbf{f}$	Global forcing vector
$\mathbf{f}^e$	Local forcing vector
N_i	Shape function of a first order triangular element
Ω	Entire problem domain
Ω^e	Domain of a single triangular element
R	Residual
ω	Arbitrary weighting function

The Newton-Raphson method

$\mathbf{J}$	Global Jacobian matrix
$\mathbf{J}^e$	Local Jacobian matrix

Air-gap elements

$\mathbf{K}^\varepsilon$	Local stiffness matrix of an air-gap element
$\mathbf{u}^\varepsilon$	Local vector of (unknown) nodal vector potentials for an air-gap element
Ω^ε	Domain of an air-gap element
α^ε	Shape function of an air-gap element
z_0	Spatial period of an axisymmetric air-gap element
λ_n	Eigenvalues

Bessel functions

I_0	Modified Bessel function of the first kind of order zero
I_1	Modified Bessel function of the first kind of order one
K_0	Modified Bessel function of the second kind of order zero
K_1	Modified Bessel function of the second kind of order one

Part I

Theory and Implementation

Chapter 1

Introduction

1.1 Electrical machine design challenges

Ever since the earliest electrical machines were built in the 1800's, inventors and engineers have busied themselves improving the designs of electrical machines. This work, which is concerned with the design of optimal electrical machines, is another drop in that ocean. It was undertaken with the renewable energy and transportation sectors and their challenges in mind, but – just as a hammer can be used for many things – this work can be just as applicable to various other electrical machines. In the following paragraphs, the design challenges of some modern electrical machines will be discussed.

Many applications, specifically many renewable energy applications require special machines that are well adapted to alternative forms of mechanical power. Some proposed wave energy systems, for example, require linear machines operating at very slow speeds. These typically have a low power density and the forces involved are very large. Direct drive machines have also become popular in wind turbines, eliminating the need for a gearbox – a component that typically requires regular maintenance. These machines also operate at relatively low speeds and the mechanical input power can be fluctuating in nature. Stirling engine applications, on the other hand, require short stroke linear machines operating at a higher frequency, typically in the order of 50 Hz. These machines must also meet other constraints on the total translator mass and force over the range of the stroke.

Electric vehicles hold the promise of cleaner, more efficient transportation and although electrical machines are not necessarily the most prominent impediment to the widespread use of these vehicles, they remain a critical component with strict design constraints. For example, the electrical machines used in the hub of a wheel have stringent volume constraints.

Efficiency has always been an important consideration, and even more so in the present day where we as the human race are growing more aware of the negative impacts of some of our activities. Although designing electrical machines for high efficiency has been quite possible for some time, the challenge lies in finding more cost effective ways of achieving this.

The cost of electrical machines have a significant impact on the total cost of electricity because these machines are critical components in the electricity generation process. Designing machines that are inexpensive can contribute towards making renewable energy generation financially more viable and reducing the negative impacts of our power systems.

Even from this brief discussion, it is clear that there are many things to consider when designing electrical machines. In order to meet all the requirements on electrical machines, the ability to accurately model their performance is of vital importance. Better modelling can lead to better designs.

1.2 Finite element analysis

The most popular technique currently available for the modelling of electrical machines is finite element analysis. This method allows many machine parameters, such as torque, power and efficiency to be calculated accurately through direct evaluation of the magnetic field in the domain of the machine and allows designs to be evaluated for many different criteria.

1.2.1 A short history of the finite element method

This account of the history of the development of the finite element method and its adoption for the solution of electromagnetic problems is partially derived from those given by Binns et al. [3] and Huebner et al. [16].

Finite element concepts were developed independently in different disciplines where problems involving differential equations with spatial variables and complex geometries arise. Here follows a time line highlighting a few milestones in the evolution of the finite element method and its application in the modelling of electrical machines.

1941 Hrenikoff [15] solves problems in elasticity by the framework method. Using this method a continuum structure is represented by a finite number of simple interconnected elements. The resulting system of equations could then be solved as if it were an ordinary truss problem.

1956 Turner et al. [27] solves plane stress problems by subdividing the problem domain into triangular elements. The direct stiffness method was introduced whereby the global stiffness matrix is assembled from the stiffness matrices of the individual triangular elements.

1960 Clough [5] coins the name *Finite Element Method* in a paper on plane stress analysis.

1963 Winslow [31] used a discretisation scheme based on an irregular mesh of triangles for the solution of electromagnetic field problems. His approach was equivalent to the finite element method and his work represents one of the first applications of the method to the solution of electromagnetic fields.

1969 Silvester [23] used higher order elements for the solution of elliptic partial differential equations.

1970 Chari and Silvester [4] were the first to use the finite element method in the analysis of electrical machines.

1979 - 1983 The use of the finite element method in the field of electromagnetics has become well established. Simkin and Trowbridge [24] presented work on the solution of static fields in three dimensions and Emson and Simkin [7] presented a method for the solution of three-dimensional eddy current problems.

1982 Time-stepped finite element simulations of electrical machines require special methods to facilitate movement of the rotor. Abdel-Razek et al. [2] introduced the air-gap element for the dynamic analysis of electrical machines.

1985 Davat et al. [6] introduced the moving band technique for the facilitation of rotor movement.

1990 The sliding surface technique for finite element simulations with moving components was introduced by Roger et al. [21].

More recent developments are largely beyond the scope of this thesis, but a few capabilities are highlighted in section 1.2.3.

1.2.2 The finite element method: basic theory

Here follows an introductory discussion on the finite element method, highlighting the fundamental concepts as is applicable to the modelling of electrical machines. A more in depth discussion can be found in Binns et al. [3].

In order to accurately calculate machine parameters, the magnetic field inside a machine must be solved. The equations that govern this field are Maxwell's equations. In the magnetostatic case which is considered in this thesis, the problem of solving the magnetic field can be reduced to solving

$$\nabla \times \frac{1}{\mu}(\nabla \times \mathbf{A}) = \mathbf{J} \tag{1.1}$$

with $\mathbf{A}$ the magnetic vector potential and $\mathbf{J}$ the current density [13, pp. 310-313 with the adaption for relative permeability]. In the two-dimensional Cartesian case $\mathbf{A}$ and $\mathbf{J}$ only have components in the z-direction and (1.1) reduces to

$$\nabla \cdot \frac{1}{\mu}\nabla A_z = -J_z \tag{1.2}$$

Using the finite element method the problem domain is broken up into small elements, usually triangles, as is illustrated in figure 1.1. On these small elements the unknown function $A_z(x, y)$ is then approximated by

$$A_z(x, y) = \alpha_1 + \alpha_2 x + \alpha_3 y = N_1 u_1^e + N_2 u_2^e + N_3 u_3^e \tag{1.3}$$

where the N_i are shape functions and the u_i^e are the (unknown) nodal values of the vector potential at the vertices of the triangle.

A general way of obtaining a solution to (1.2) is to integrate the residual,

$$R = \nabla \cdot \frac{1}{\mu}\nabla A_z + J_z \tag{1.4}$$

multiplied by a weighting function over the problem domain Ω, namely

$$\int \omega_i R \mathrm{d}\Omega = 0 \tag{1.5}$$

It can be seen that if (1.5) is satisfied for any ω_i then R must be zero over the entire problem domain and (1.2) is satisfied. This method is known as the method of weighted residuals. In practice it is neither possible nor necessary to test all ω_i. A finite number will suffice. Thus, (1.5) becomes

$$\int_\Omega \sum_i \omega_i R \mathrm{d}\Omega = 0 \tag{1.6}$$

Because the problem domain is broken up into small triangular elements, the integral over the problem domain in (1.6) is replaced by the sum of the integrals over the elements,

$$\sum_{n=1}^{N} \int_{\Omega^e} \sum_i \omega_i R \mathrm{d}\Omega^e = 0 \tag{1.7}$$

where there are N elements and Ω^e is the domain of a single triangular element. Considering only a single element and substituting (1.4) in (1.7), one obtains

$$\int_{\Omega^e} \sum_i \omega_i (\nabla \cdot \frac{1}{\mu}\nabla A_z + J_z) \mathrm{d}\Omega^e = 0 \tag{1.8}$$

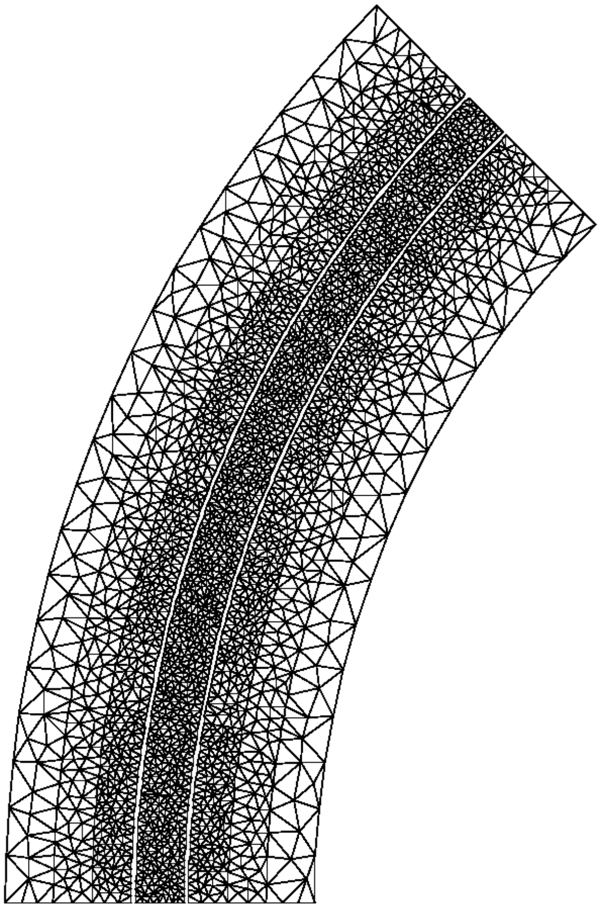


Figure 1.1: A typical finite element mesh.

Using Green's theorem this can be written as

$$\int_{\Omega^e} \sum_i \nabla \omega_i \frac{1}{\mu} \nabla A_z \mathrm{d}\Omega^e - \int_{\Omega^e} \sum_i \omega_i J_z \mathrm{d}\Omega^e - \int_{\Gamma_e} \sum_i \omega_i \frac{1}{\mu} \nabla A_z \mathrm{d}\Gamma_e = 0 \tag{1.9}$$

The third term in the equation above is zero for the type of problem considered in this thesis and will be ignored from here on. Substituting (1.3) into (1.9) one obtains

$$\int_{\Omega^e} \frac{1}{\mu} \sum_i \left(\nabla \omega_i \sum_{j=1}^{3} \nabla N_j u_j^e \right) \mathrm{d}\Omega^e - \int_{\Omega^e} \sum_i \omega_i J_z \mathrm{d}\Omega^e = 0 \tag{1.10}$$

At this point it is noted that if the number of weighting functions used is equal to the number of unknowns, u_j^e, (1.10) forms a set of linear equations and can be written as

$$\mathbf{K}^e \mathbf{u}^e = \mathbf{f}^e \tag{1.11}$$

with $\mathbf{K}^e$ denoting the local element stiffness matrix[1]. The superscripts indicate local element systems. If the weighting functions are chosen, according to the Galerkin method, as the shape functions, $\omega_i = N_i$, then

$$K_{ij}^e = \int_{\Omega^e} \frac{1}{\mu} \nabla N_i \nabla N_j \mathrm{d}\Omega^e \qquad f_i^e = \int_{\Omega^e} N_i J_z \mathrm{d}\Omega^e \tag{1.12}$$

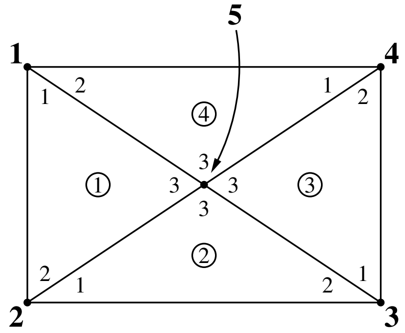


Figure 1.2: A portion of a finite element mesh.

Now, to assemble the global stiffness matrix, **K**, refer to figure 1.2 where the large bold numbers indicate global node numbers, the small numbers indicate local node numbers and the circled numbers indicate element numbers. To calculate a term, K_{ij} of the global system matrix, terms from different local system matrices may have to be summed. For example, K_{25} and f_2 is calculated as

$$K_{25} = K_{13}^2 + K_{23}^1 \qquad f_2 = f_2^1 + f_1^2 \tag{1.13}$$

where the superscripts indicate local element systems. The term, K_{55}, is calculated as

$$K_{55} = K_{33}^1 + K_{33}^2 + K_{33}^3 + K_{33}^4 \tag{1.14}$$

Application of this procedure leads to the global system

$$\mathbf{Ku} = \mathbf{f} \tag{1.15}$$

which can be solved for the vector potential at every node **u**.

1.2.3 The capabilities of modern finite element analysis

The previous section described how the finite element method could be used to solve the magnetic vector potential A_z in a two-dimensional linear magnetostatic problem. Useful as this may be, it is by no means the limit of what modern day finite element analysis can do. As has been indicated, the method is extendible to three dimensions, capable of simulating non-linear magnetic materials and solving time-dependent fields with eddy currents and motional effects. More recently the simulation of electrical machines coupled to external circuits has been approached by coupling of the field and circuit equations [32].

All of this functionality does, however, come at a high computational cost. Therefore, a trade-off exists between simulation accuracy and computational time, which is still

[1]The term *stiffness matrix* is inherited from structural finite element analysis, where the method was first used. In the present context, the matrix does, in fact, have nothing to do with stiffness.

an important consideration even with modern computers. It is often possible to obtain reasonable accuracy with a simplified model that reduces the time required to obtain a solution.

Apart from being computationally expensive, finite element analysis is still not perfect. Accurate calculation of hysteresis-losses, specifically, is not well established. Recent papers on this topic include [19] and [18].

1.3 Optimisation of electrical machines

Although it is possible to optimise machine designs based on analytical solutions of machine parameters, this method is usually less accurate and not as powerful as multivariable optimisation using finite element analysis. The latter process, which is focused on in this thesis, is illustrated in figure 1.3. The idea is to define the optimisation problem as follows,

$$\text{Minimise} \; : \; F(\mathbf{X}) \tag{1.16}$$

$$\begin{aligned} \text{Subject to} \; : \; & G_1(\mathbf{X}) > g_1 \\ & \vdots \\ & G_k(\mathbf{X}) > g_k \end{aligned} \tag{1.17}$$

$$X_{li} <= X_i <= X_{ui} \tag{1.18}$$

where $\mathbf{X}$ is the vector of design variables. Equation 1.16 is the objective function and (1.17) represents a total of k constraints which are also functions of $\mathbf{X}$. The search space is defined by lower and upper bounds on each design variable, X_{li} and X_{ui}, as in (1.18). Referring to figure 1.3, the finite element simulation is responsible for evaluating the objective function and constraints for a vector of design variables. The optimiser's task is to find the optimal choice of design variables. In order to do this, it is necessary to evaluate the objective function and the constraints many times.

Considering this process, it is clear why it is necessary to minimise the computational time of finite element simulations: A single optimisation process requires many simulations to be run. Depending on the complexity of the problem, this means that the effect of computational efficiency for a single optimisation is not measured in seconds or in minutes, but in hours, days or even weeks!

Just as important is the number of function evaluations needed by the optimisation algorithm to find the optimal design. There are many different numerical optimisation algorithms to choose from that work in very different ways (see for example the book by Vanderplaats [28]). An incorrect choice of algorithm may result in sub-optimal designs or unnecessary time being spent on optimisation.

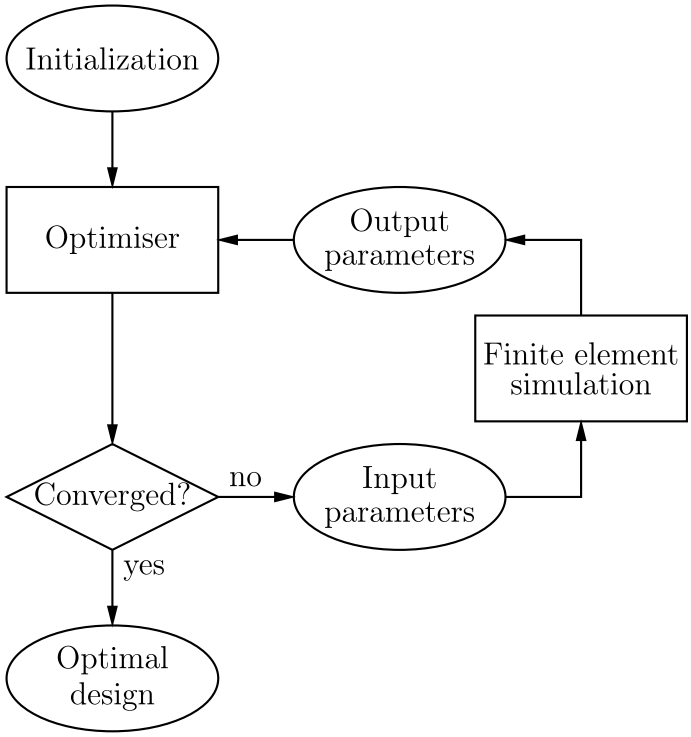


Figure 1.3: Design optimisation using finite element analysis.

1.4 Overview of this work

1.4.1 Motivation

In the past, a custom finite element program, known as the *Cambridge* package among its users, has been used with good success by members of the Electrical Machines Group at the University of Stellenbosch. Because of the importance of finite element analysis in electrical machine design and the versatility offered by having the source code of a finite element program available, this program was deemed a significant asset. Unfortunately, there were also a couple of negative aspects to the implementation such as the old Fortran 77 standard that it was written in, the difficulty of generating a mesh and the overarching structure of the program. It was considered a worthwhile undertaking to sort out these issues and make some improvements that would allow the continued use of this program.

1.4.2 Goal of this work

The goal of this work was to transform the original finite element program into a more powerful and usable tool, making it more competitive with commercially available finite element simulation packages for certain classes of problems. The classes of problems targeted in this work were two-dimensional magnetostatic problems with non-linear magnetic materials. These classes, illustrated in figures 1.4 to 1.6, can be defined as follows:

1. 2D rotating machines
2. 2D flat linear machines
3. 2D axisymmetric or tubular linear machines

The first two classes require the solution of (1.1) in Cartesian coordinates,

$$\frac{\partial^2 A_z}{\partial x^2} + \frac{\partial^2 A_z}{\partial y^2} = -\mu J_z \tag{1.19}$$

and the third class requires the solution of (1.1) in cylindrical coordinates

$$\frac{\partial^2 A_\phi}{\partial z^2} + \frac{\partial^2 A_\phi}{\partial r^2} + \frac{1}{r}\frac{\partial A_\phi}{\partial r} - \frac{1}{r^2}A_\phi = -\mu J_\phi \tag{1.20}$$

where μ is a function of the magnetic field strength, defined by a single valued B-H curve, and the current densities J_z and J_ϕ are prescribed. These three cases cover all problems that can be simplified to two dimensions.

The original program was capable of simulating class I and II problems, but with the limitation that only a single air-gap could be modelled. The simulation of class III problems was not implemented at all. Thus, for class I and II problems the program needed to be extended to simulate problems with multiple air-gaps while all the calculations needed to be adapted for the case of class III problems.

Another requirement was that the developed finite element program should be easy to use within different optimisation environments and that some optimisation methods should be investigated in order to make a recommendation regarding the best methods. If a robust, tried and tested, environment for the optimisation of electrical machines could be created, it could be a very powerful tool.

1.4.3 Layout of this thesis

Chapter 2: This chapter gives a detailed description of the original *Cambridge* package as received by the author. The program's capabilities are discussed and details about several important calculations are given. Strengths and weaknesses are also highlighted.

Chapter 3: This chapter discusses the process of restructuring the original program and gives details about several improvements that were made.

Chapter 4: Attention is given to issues regarding relative movement in the finite element mesh, including the expansion of the original program to allow simulation of machines with multiple air-gaps using the air-gap element method.

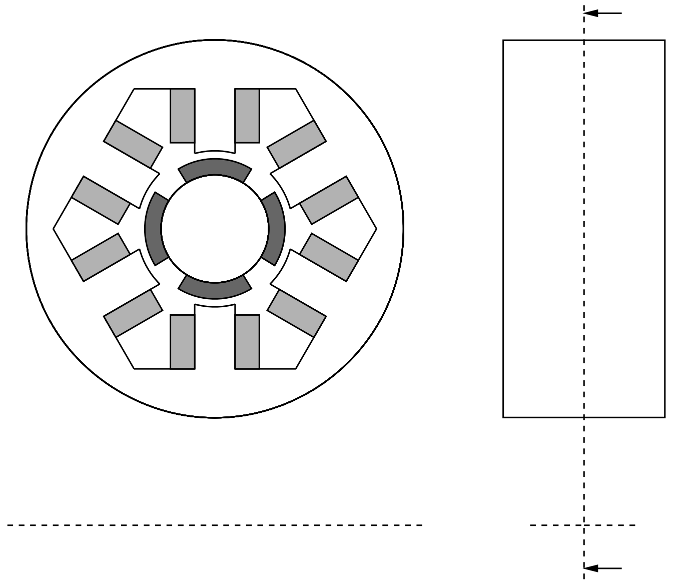


Figure 1.4: Problem class I: 2D rotating machines.

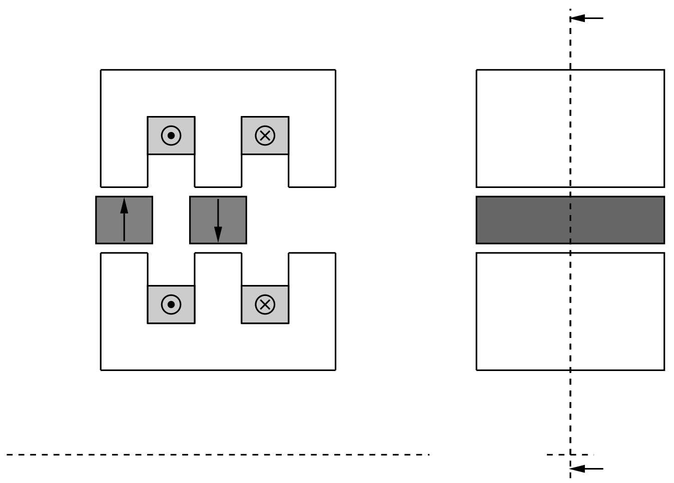


Figure 1.5: Problem class II: 2D flat linear machines.

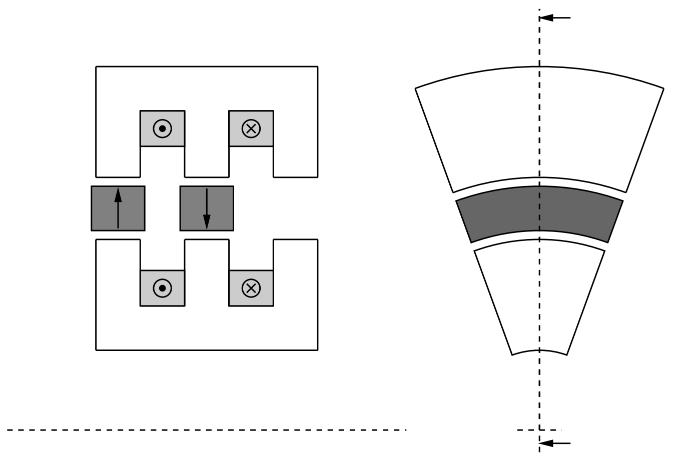


Figure 1.6: Problem class III: 2D axisymmetric linear machines.

Chapter 5: This chapter deals with all issues relevant to the simulation of class III problems, including the derivation of the axisymmetric air-gap element.

Chapter 6: In this chapter, the focus is on the optimisation of electrical machines using finite element analysis. The capability to exploit parallel processors to speed the optimisation process, is discussed.

Chapter 7: This chapter illustrates the use of the finite element program through the presentation of a few case studies. The case studies are chosen so as to illustrate the different types of machines that can be simulated using the program and to verify the accuracy – both in derivation and implementation – of calculations presented in previous chapters.

Chapter 8: In this chapter, a summary of goals accomplished in this work is given. Recommendations regarding future development of the finite element package are also made.

Chapter 2

Overview of the original program

2.1 Introduction

In this chapter, an overview is given of the original finite element program, the *Cambridge* package, as it was when first received by the author. This is done to provide the reader with a clear picture of the functionality that was implemented prior to this work and also to give credit to previous contributors where it is due. On the other hand, the inadequacies of the original program are also discussed, highlighting the significance of the improvements that were made. The overview is detailed in the sense that all of the major components of the program are discussed without going into the finer points of all the subsystems – many of which represent entire fields of their own.

Almost all of the calculations that are documented in this chapter were not available to the author at the beginning of this work. In many cases the author came to understand these calculations from intense study of the code, the references therein and other relevant literature. Indeed, the task of maintaining the program may have been much simpler if more elaborate documentation of the functioning of the program existed. Although this chapter can not completely satisfy this need, it aims to serve as a good starting point.

The chapter starts off by discussing the structure of the original program in section 2.2, explaining the inter-dependencies of the different components. This is followed by a critical assessment of the original program in section 2.3. Thereafter, some of the data structures that are used to store information related to the mesh and the system equation is discussed in section 2.4. These data structures are necessary to improve the storage and computational efficiency of the program. Next, the original method of constructing a mesh is examined in section 2.5. This method, although simple in principle, was not really adequate for the purpose of this program. In section 2.6, the special air-gap elements used by the program are introduced. These air-gap elements facilitate movement in the mesh and allow accurate force calculation using the Maxwell stress tensor method. The basic theory of air-gap elements is presented. The method of solving (1.15) when $\mathbf{K}$ is a non-linear function of $\mathbf{u}$ is discussed in section 2.7. Finally, in sections 2.8 and 2.9, two important post-processing calculations, the force and flux linkage calculations, are presented in detail.

2.2 Original structure

2.2.1 Overview

First of all, it must be explained that the original program was not a single program. Two programs existed, one for rotating machines (class I) and one for linear machines (class II). Originally there was no capability to simulate class III problems. An important inherent problem with this approach is that it necessitates duplication of large portions of code. This approach leads to code that is more difficult to maintain because improvements made in one version is not automatically incorporated into the other version – an inconvenience that becomes especially significant in large projects such as the present one. The alternative is to have a single program with no duplication of code, but with added logic to allow the simulation of different types of machines.

A graphical representation of the original program illustrating the major components from the user's perspective is shown in figure 2.1. File dependencies are also illustrated. The solid arrowheads indicate a *provides* or *implements* relationship while the line arrowheads indicate a dependency relationship. The user's work in this scenario is effectively distributed over six areas and the components have many file dependencies.

The `eesolv` component is the main component called by the encompassing optimisation routine, which can also be used to run a single simulation. The `ee_as` component is where a mesh is assembled from smaller mesh parts. The user is responsible for drawing a machine in the `ee_as` component and its sub-components. The output of this process is a .pol file (pol for polygon). The polygon file is used by the `ee_pmesh` component to generate a mesh as is discussed in section 2.5. The output of this process is a .fpl file (fpl for field plot). The information stored in this file is used extensively throughout the rest of the program. In the `ee_pre` component the air-gap elements are set up and the system equation is prepared. This equation is then solved and post-processing calculations are made in the `eesolv` component.

The `ee_as` and `ee_pre` components require a definition file, rotor lamination file, stator lamination file, rotor slot file and stator slot file to function correctly, some of which the user must prepare and other which is generated by the `eesolv` component, although the

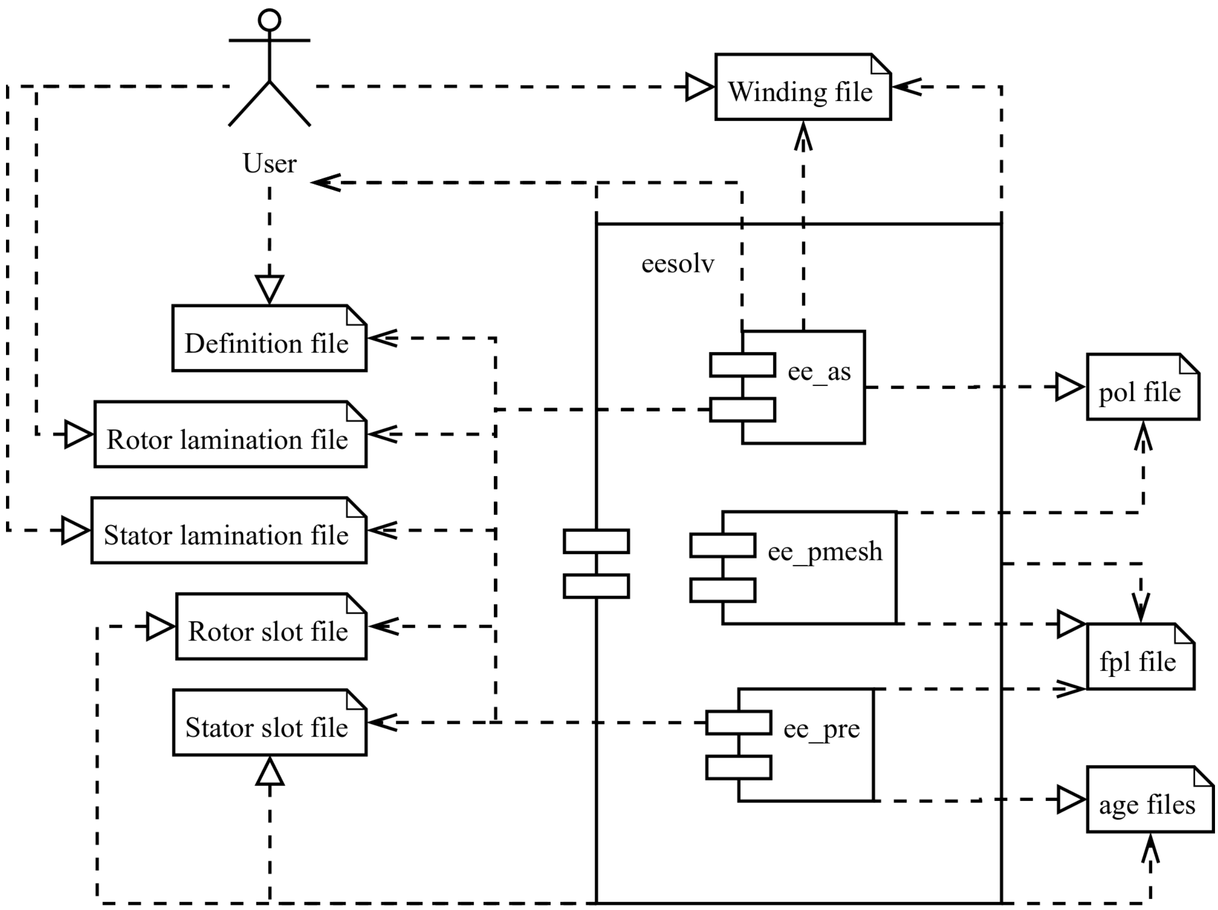


Figure 2.1: The structure of the original program from the user's perspective.

user must also modify this code for different machines. The `ee_as` and `eesolv` components also require a winding file.

Listing 2.1 describes the high-level structure of the two original programs. In the following paragraphs, the functions shown in these listings will be discussed in detail.

2.2.2 Input files

Before running a simulation, the user had to set up a couple of input files correctly. These files were the definition file, the rotor and stator lamination files, the winding file and the B-H file. The definition file contained the file names of the lamination files and the winding file, making it unnecessary to hard code these. The lamination files contained information on the total number of slots, the number of modelled slots, slot positioning as well as references to slot files which were generated by the program. Both lamination files also contained the B-H file name, although these file names could not be different. The B-H file, not shown in figure 2.1, mapped magnetic field intensity to flux density for the non-linear material used in the simulation. The user provided the B-H file.

```
! User must initialize:
! definition file
! lamination files
! winding file, bh file

program eeoptmb
  ! Initialize optimisation (xpar)
  call eesolv(xpar, ypar)
  ! Optimisation calculations
  ! eesolv is called repeatedly
end program

subroutine eesolv(xpar, ypar)
  ! Machine specific parameters are initialized here
  call save_rslot(...)          ! Write rotor slot file
  call save_sslot(...)          ! Write stator slot file
  call ee_as(def_file,...)      ! Generate .pol file
  call ee_pmesh(...)            ! Generate .fpl file
  call ee_pre(...)              ! Generate age files
  call sreadmesh(...)           ! Read .fpl file
  call read_sw(...)             ! Read winding file

  ! Read air-gap element data from age files
  ! Solve
  ! Post-process
end subroutine

subroutine ee_as(...)
  ! Read definition file
  ! Read winding file
  ! Read stator lamination file
  call ee_pol(slot_file, pol_file, ...)
  ! Read rotor lamination file
  call ee_pol(slot_file, pol_file, ...)
  ! Generate whole machine .pol file
  ! from .pol slot files
end subroutine

subroutine ee_pol(slot_file, pol_file, ...)
  ! Read slot file (different file formats)
  ! Generate pol_file
end subroutine
```

```
subroutine ee_pmesh(...)
  ! Read .pol file
  ! Generate mesh
  ! Write .fpl file
end subroutine

subroutine ee_pre(...)
  ! Read definition file
  ! Read stator and rotor lamination files
  ! Read stator and rotor slot files
  call ex_age_nds(...)
  call mk_age_t(...)
  call pre_all(...)
end subroutine ee_pre(...)

subroutine ex_age_nds(...)
  ! Read .fpl file
  ! Generate age.age
end subroutine

subroutine mk_age_t(...)
  ! Read age.age
  ! Generate age.res, age.rtm
end subroutine

subroutine pre_all(...)
  ! Read .fpl file
  ! Setup boundary conditions
  ! Other preprocessing (requires age.age)
  ! Generate age.ren
end subroutine
```

Listing 2.1: Structure of the original program.

2.2.3 program `eeoptmb`

This is the main program. It can be configured to run either a single simulation or to perform an optimisation. In both cases, the array of design variables, `xpar`, is initialized and subroutine `eesolv` is called to simulate the design. When doing an optimisation, the array of simulation results, `ypar`, is used to determine the next set of design variables to be evaluated. The optimisation algorithm used is Powell's method.

2.2.4 subroutine `eesolv`

This subroutine accepts an array of design variables, `xpar`, as input, simulates the design and returns an array of results, `ypar`.

The subroutines `save_rslot` and `save_sslot` generated the rotor and stator slot files respectively. These slot files contained information on the total number of slots as well as slot dimensions. All this data was hard coded in the `eesolv` subroutine. There was some duplication of data between the slot files generated by the program and the lamination files generated by the user.

The subroutine `ee_as` is responsible for generating a polygon file. This file contains a list of polygons and nodes that describe the geometry of the machine to be simulated. It is the input to the `ee_pmesh` subroutine which generates the finite element mesh data structures contained in a field plot file.

The generated mesh consists of two distinct parts, the stator and the rotor with a gap between them. The subroutine `ee_pre` performs all the calculations related to the Air-Gap Element (AGE) or Cartesian Air-Gap Element (CAGE), depending on the version of the program. These air-gap elements connect the two parts of the mesh together and allow movement between the two parts to be handled easily. This subroutine also generates other data structures needed by `eesolv`.

Next, the mesh data and the winding data is read from the field plot and winding files respectively and all the generated air-gap element data is read. Then the system matrix equation is set up and solved. This process is complex and will not be discussed in detail here. After the system equation has been solved and the magnetic vector potential is known at every node, some post-processing calculations can be made. These typically include flux linkage, torque or force, terminal voltage, output power, efficiency etc. The critical results are copied to the array `ypar` and the subroutine terminates.

2.2.5 subroutine `ee_as`

From the definition file, winding file, lamination files and slot files, this subroutine generates a polygon file describing the geometry of the machine. It typically calls the subroutine `ee_pol` multiple times with different slot files in order to generate the different slots of the machine in polygon file format. After all the slot polygon files have been generated, it combines them into a single polygon file representing the entire machine, including the stator and rotor.

2.2.6 subroutine `ee_pol`

This subroutine generates polygon files for individual slots. This is where the user "draws" the machine. The subroutine reads the slot file passed to it, identifies the type of slot and then runs the appropriate sections of code to generate the correct polygon structure. This subroutine is customized for every different machine.

2.2.7 subroutine `ee_pmesh`

This subroutine takes the polygon file representing the machine as input and generates the field plot file containing the finite element mesh data structures as output. It does this by adding a single node in the centre of every polygon in the polygon file and connecting this node to the nodes on the edges of the polygon to form triangles. After this process is complete it employs a Delaunay algorithm to optimise the mesh. See section 2.5 for details.

2.2.8 subroutine `ee_pre`

This subroutine handles all calculations related to the air-gap elements. It is also responsible for applying the appropriate boundary conditions as well as generating other data structures needed by the solver (see section 2.4). All the data generated by this subroutine is stored in four files, namely `age.age`, `age.res`, `age.rtm` and `age.ren`.

2.3 Criticism of the original program

Although the program in its original forms was already considered to be a powerful tool in the design of electrical machines and a useful alternative to commercially available finite element simulation packages, there were a couple of negative aspects to the program that warranted some restructuring. Note that the criticism offered in this section is criticism on how the original program performed the tasks that it could perform. Specifically, no mention is made of the limitations of the program regarding problems with multiple air-gaps and class III problems. The most pressing negative points are listed here:

- The program was written in Fortran 77, an old standard upon which many improvements have been made in more recent years.
- The program performed many unnecessary file operations related to simulation inputs and internally generated data. Some of these operations could be ascribed to

the fact that the code was written in an era when memory usage was critical and had to be minimised wherever possible. However, on modern computers these operations are unnecessary and only have a negative impact on performance. Depending on the size of the model, some of the AGE files specifically could be very large (in excess of 10MB). It seemed very inefficient to write all this data to a file, only to read it back a few steps later.

Not only did these file operations have a negative impact on performance, but it also complicated the task of following the work flow of the program.

- The program was structured in such a way that the input describing a machine was split across multiple files which were hard to interpret. Duplication of data also occurred in some places.

- Machines were drawn in the correct format for this program using a scripting approach. This in itself was not a problem and is in fact desirable for optimisation purposes. However, drawing machines was a tedious process because of a lack of convenience functions and no easily accessible graphical output capabilities. Varying the coarseness of a mesh was just as tedious a process because it was almost equivalent to constructing an entirely new drawing. This was a major obstacle in making the program easy to work with.

- The mesh generation algorithm makes no provision for dynamically adjusting the number of nodes in different areas of the mesh. This can be a problem when an optimisation process evaluates a design with dimensions that are very different from the original design upon which the user based the mesh density. In other words, there was no way to specify an absolute maximum size for a triangular element. The number of nodes where fixed in every area of the machine, regardless of the actual size of those areas.

- Magnets were modelled using current sheets. (see section 3.5) Although this a nice simple method for modelling the effects of permanent magnetisation, it requires more effort from the user when constructing the mesh and is not capable of modelling magnets with complex shapes. Only simple rectangles and arc sections could be modelled using this method.

- Another source of inconvenience was the fact that there was no formal distinction between the core program and machine specific code. This complicated the task of simulating more than one machine with the program because a completely different version of the program was needed for every machine and it was not clear which parts of the code was generic and which parts were machine specific. This further complicated the task of keeping all versions of the program up to date.

- Many of the working arrays in the program were declared to have a fixed length that could not adjusted without modifying large portions of variable declarations. In effect, this limited the size of problem that could be solved with the original program.

It was the aim of this work to address these issues by completely restructuring the original versions of the program. See section 3.2 for a discussion on the restructuring process.

2.4 Data structures and profile reduction

In this section a couple of important arrays used in the program are discussed with the aid of a simple example. These arrays are used to store information that is used in the process of assembling and solving the global system equation (1.15). A naive implementation of (1.15) may lead to a scheme where the global stiffness matrix $\mathbf{K}$ is stored in a simple two-dimensional array and the nodes in $\mathbf{u}$ are ordered according to the order of appearance in the field plot file. Such a scheme would, however, lead to inefficient use of memory storage and sub-optimal performance in terms of computational time[1]. Improvements are possible because the stiffness matrix is sparse and most entries do not need to be stored. This fact is best exploited when the entries in the system equation are ordered in a clever way, resulting in a stiffness matrix with a smaller profile. Thus, the data structures discussed in this section are used to store the system equation in such a way as to improve efficiency in terms of storage and computational time. The use of the profile reduction algorithm discussed in section 2.4.2 enhances this effect.

The preprocessor is responsible for generating most of the data discussed in this section. The example model was specifically chosen small enough (only 54 nodes) so that the entire mesh could be inspected and the stiffness matrix was of managable size (19 x 19). This makes it possible to wrap one's mind around the problem which is typically much larger for useful problems.

Figure 2.2 shows the mesh of this example with no profile reduction performed. It is a simple magnet surrounded by air. In the original version of the program it was not possible to model magnets as simply as this. Magnets were modelled using current sheets (see section 3.5). However, the method of modelling magnets has no impact on this discussion. The meaning of the numbers in the figure will become clear in the following discussion of important arrays. The stiffness matrix for this example is shown in figure 2.3. Note that the element K_{ij} is written as K^i_j for the sake of compactness. This matrix

[1] Although the method used in the original program is certainly better than the simple implementation described here, it is not claimed that this method is optimal

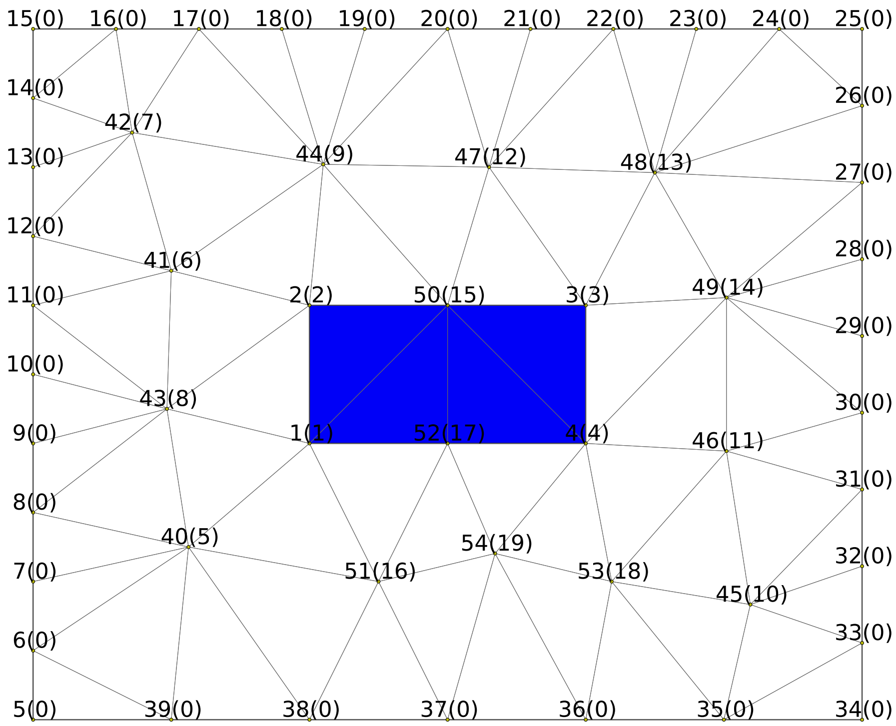


Figure 2.2: Mesh plot showing node numbers and variable numbers with no profile reduction.

was constructed by direct application of the procedure discussed at the end of section 1.2.2. The symmetric nature of this matrix results directly from the symmetry in (1.12) ($K^e_{ij} = K^e_{ji}$). Only the framed part of the matrix is actually stored in memory. Zeros are indicated by dots to improve legibility.

2.4.1 Description of arrays

In this section, the most important arrays that are used to prepare and store the stiffness matrix is discussed in some detail. Throughout the discussion, reference is made to the example of figure 2.2 to illustrate the use of the arrays in a practical way.

`integer nd(:)`

This array, the nodal directory, maps node numbers to unknown variables in the system equation. Referring to figure 2.2, the node numbers are the numbers above every node and the variable numbers are displayed in brackets. In this context, a variable corresponds to a vector potential at a node that is not constrained to a specific value by a boundary condition. For example, node number one is mapped to variable number one, node number two to variable number two and node number forty to variable number five. Note that all the nodes on the edge of the model are mapped to variable number zero, meaning that they are not variables. This is because a Dirichlet boundary condition ($A_z = 0$) was applied on the outer edge

$$
\begin{array}{ccccccccccccccccccc}
K_1^1 & K_2^1 & . & . & K_5^1 & . & . & K_8^1 & . & . & . & . & . & . & K_{15}^1 & K_{16}^1 & K_{17}^1 & . & . \\
K_2^1 & K_2^2 & . & . & . & K_6^2 & . & K_8^2 & K_9^2 & . & . & . & . & . & K_{15}^2 & . & . & . & . \\
. & . & K_3^3 & K_4^3 & . & . & . & . & . & . & . & K_{12}^3 & K_{13}^3 & K_{14}^3 & K_{15}^3 & . & . & . & . \\
. & . & K_4^3 & K_4^4 & . & . & . & . & . & . & K_{11}^4 & . & . & K_{14}^4 & K_{15}^4 & . & K_{17}^4 & K_{18}^4 & K_{19}^4 \\
K_5^1 & . & . & . & K_5^5 & . & . & K_8^5 & . & . & . & . & . & . & . & K_{16}^5 & . & . & . \\
. & K_6^3 & . & . & . & K_6^6 & K_7^6 & K_8^6 & K_9^6 & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & K_7^6 & K_7^7 & . & K_9^7 & . & . & . & . & . & . & . & . & . & . \\
K_8^1 & K_8^2 & . & . & K_8^5 & K_8^6 & . & K_8^8 & . & . & . & . & . & . & . & . & . & . & . \\
. & K_9^2 & . & . & . & K_9^6 & K_9^7 & . & K_9^9 & . & . & K_{12}^9 & . & . & K_{15}^9 & . & . & . & . \\
. & . & . & . & . & . & . & . & . & K_{10}^{10} & K_{11}^{10} & . & . & . & . & . & . & K_{18}^{10} & . \\
. & . & . & K_{11}^4 & . & . & . & . & . & K_{11}^{10} & K_{11}^{11} & . & . & K_{14}^{11} & . & . & . & K_{18}^{11} & . \\
. & . & K_{12}^3 & . & . & . & . & . & K_{12}^9 & . & . & K_{12}^{12} & K_{13}^{12} & . & K_{15}^{12} & . & . & . & . \\
. & . & K_{13}^3 & . & . & . & . & . & . & . & . & K_{13}^{12} & K_{13}^{13} & K_{14}^{13} & . & . & . & . & . \\
. & . & K_{14}^3 & K_{14}^4 & . & . & . & . & . & . & K_{14}^{11} & . & K_{14}^{13} & K_{14}^{14} & . & . & . & . & . \\
K_{15}^1 & K_{15}^2 & K_{15}^3 & K_{15}^4 & . & . & . & . & K_{15}^9 & . & . & K_{15}^{12} & . & . & K_{15}^{15} & . & K_{17}^{15} & . & . \\
K_{16}^1 & . & . & . & K_{16}^5 & . & . & . & . & . & . & . & . & . & . & K_{16}^{16} & K_{17}^{16} & . & K_{19}^{16} \\
K_{17}^1 & . & . & K_{17}^4 & . & . & . & . & . & . & . & . & . & . & K_{17}^{15} & K_{17}^{16} & K_{17}^{17} & . & K_{19}^{17} \\
. & . & . & K_{18}^4 & . & . & . & . & . & K_{18}^{10} & K_{18}^{11} & . & . & . & . & . & . & K_{18}^{18} & K_{19}^{18} \\
. & . & . & K_{19}^4 & . & . & . & . & . & . & . & . & . & . & . & K_{19}^{16} & K_{19}^{17} & K_{19}^{18} & K_{19}^{19}
\end{array}
$$

Figure 2.3: The stiffness matrix corresponding to figure 2.2.

of the mesh which enforces a value of zero on these nodes. This type of boundary condition prevents any flux from penetrating the boundary.

`integer ndeg(:)`

This array stores the variable degree vector. The degree of a variable indicates to how many other variables it is connected. Referring to figure 2.2, the degree of variable five is three because three outgoing lines from variable five (node forty) are connected to other variables. Similarly, the degree of variable one (node one) is six. The degree of a variable is also reflected in the stiffness matrix shown in figure 2.3. Looking at the i'th row (or column), the degree of variable i is equal to the number of non-zero terms in the row (or column), excluding the term on the main diagonal. The array for this example is shown in table 2.1.

Table 2.1: The array `ndeg(:)` for this example.

i	1	2	3	4	5	6	7	...	12	13	14	15	16	17	18	19
ndeg(i)	6	5	5	7	3	4	2	...	4	3	4	7	4	5	4	4

`integer ncon(:,:)`

This array is used to keep track of which variables are connected. The first couple of entries for this example case is shown in table 2.2. The number of entries in the i'th row of the array corresponds to `ndeg(i)`. The meaning of this array can be interpreted in terms of either figure 2.2, where the i'th row of `ncon` contains the variables connected to the i'th variable, or in terms of figure 2.3, where the i'th row of the stiffness matrix has non-zero elements at the columns contained in the i'th row of `ncon` and on the diagonal.

Table 2.2: The first couple of entries in the array `ncon(:,:)` for this example.

ncon		**j**							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
i	**1**	2	15	8	5	16	17		
	2	6	8	1	15	9			
	3	13	14	12	15	4			
	4	11	18	3	15	19	14	17	
	5	16	8	1					

`double precision st(:)`

This array is used to store the framed part of the stiffness matrix shown in figure 2.3. Note that only terms above the main diagonal are stored because the matrix is symmetric. This array, along with `jdiag(:)` described next, implements a storage scheme known as jagged diagonal storage. This scheme is very space efficient for sparse banded matrices such as the system matrices produced by the program because it requires storage of almost only the non-zero terms of the matrix, stored in `st(:)`, and an index to the location of the terms on the main diagonal, stored in `jdiag(:)`. This may not be apparent looking at figure 2.3 because the matrix has a large profile. The benefits of this storage scheme will become more apparent when the profile reduction function of the preprocessor is discussed. The first couple of entries in `st(:)` for this example is shown in table 2.3. The framed part of the matrix is stored column by column.

Table 2.3: The array `st(:)` for this example.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...
st(i)	K_1^1	K_2^1	K_2^2	K_3^3	K_4^3	K_4^4	K_5^1	.	.	.	K_5^5	K_6^2	.	.	...

`integer jdiag(:)`

This array is an index to the terms on the main diagonal of the stiffness matrix stored in `st(:)`. The i'th diagonal term is stored in `st(jdiag(i))`. The position in `st(:)` of any term in the framed part of the matrix can be calculated as follows,

$$k = \texttt{jdiag}(column) + row - column \tag{2.1}$$

where *column* corresponds to i, *row* to j and the term K_j^i is then located in `st(k)`. For example, to calculate the position of the term K_6^2

$$\begin{aligned} \mathrm{k} &= \texttt{jdiag(6)} + 2 - 6 && (2.2)\\ &= 16 + 2 - 6 && (2.3)\\ &= 12 && (2.4) \end{aligned}$$

The array for this example is shown in table 2.4.

Table 2.4: The array `jdiag(:)` for this example with no profile reduction.

i	1	2	3	4	5	6	7	...	12	13	14	15	16	17	18	19
jdiag(i)	1	3	4	6	11	16	18	...	53	64	76	91	107	124	139	155

2.4.2 Preprocessor profile reduction

The profile of a matrix is defined in the following manner [12],

$$f_i = \min\{j : a_{ij} \neq 0\} \qquad (2.5)$$

$$\text{profile} = \sum (i - f_i) \qquad (2.6)$$

where f_i is the smallest index of a column containing a non-zero entry in the i'th row and the profile is calculated as the sum, over all the rows, of the difference between this index and the index of the diagonal. Referring to figure 2.3, it is clear that many zeros need to be stored because the profile of the matrix is large. The preprocessor overcomes this inefficiency by reducing the profile of the stiffness matrix using the algorithm presented by Gibbs et al. [12]. The profile is reduced by a clever choice of variable numbering. Figure 2.4 shows the variable numbers after the profile reduction operation was executed. Figure 2.5 shows the reduced stiffness matrix which can now be stored much more efficiently. The array `jdiag(:)` for the matrix in figure 2.5 is shown in table 2.5. Comparing the last entries in tables 2.4 and 2.5 which corresponds to the total size of `st(:)`, a reduction from 155 to 79 elements is observed. For larger problems, the reduction is usually even more significant.

Table 2.5: The array `jdiag(:)` for this example with profile reduction.

i	1	2	3	4	5	6	7	...	12	13	14	15	16	17	18	19
jdiag(i)	1	3	5	8	13	17	22	...	47	51	55	62	66	72	76	79

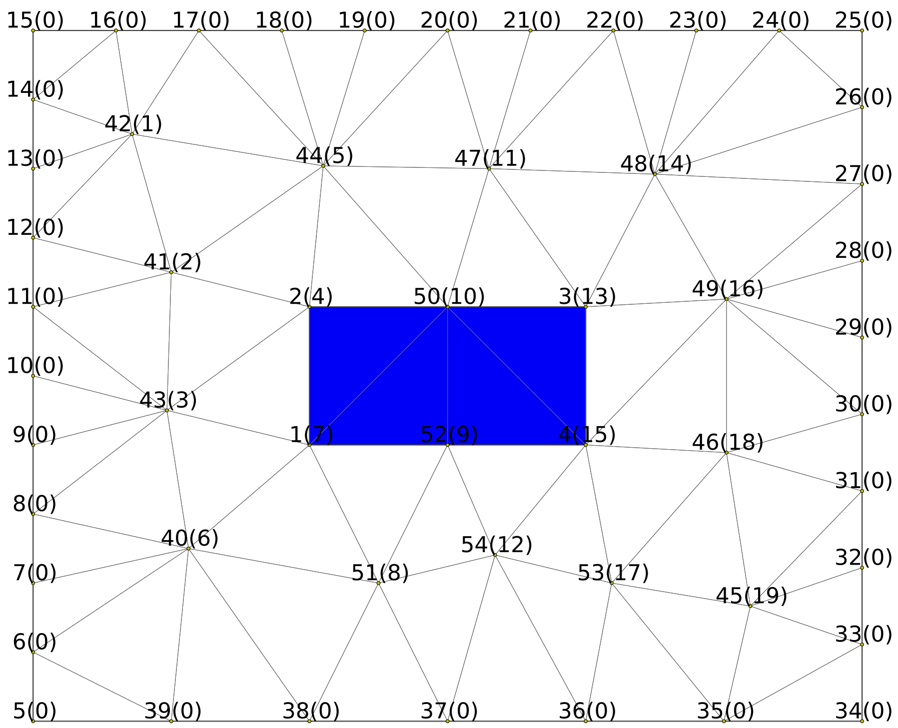


Figure 2.4: Mesh plot showing the node numbers and variable numbers after profile reduction was performed.

$$
\begin{matrix}
K_1^1 & K_2^1 & \cdot & \cdot & K_5^1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
K_2^1 & K_2^2 & K_3^2 & K_4^2 & K_5^2 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & K_3^2 & K_3^3 & K_4^3 & \cdot & K_6^3 & K_7^3 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & K_4^2 & K_4^3 & K_4^4 & K_5^4 & \cdot & K_7^4 & \cdot & \cdot & K_{10}^4 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
K_5^1 & K_5^2 & \cdot & K_5^4 & K_5^5 & \cdot & \cdot & \cdot & \cdot & K_{10}^5 & K_{11}^5 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & K_6^3 & \cdot & \cdot & K_6^6 & K_7^6 & K_8^6 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & K_7^3 & K_7^4 & \cdot & K_7^6 & K_7^7 & K_8^7 & K_9^7 & K_{10}^7 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & K_8^6 & K_8^7 & K_8^8 & K_9^8 & \cdot & \cdot & K_{12}^8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_9^7 & K_9^8 & K_9^9 & K_{10}^9 & \cdot & K_{12}^9 & \cdot & \cdot & K_{15}^9 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & K_{10}^4 & K_{10}^5 & \cdot & K_{10}^7 & \cdot & K_{10}^9 & K_{10}^{10} & K_{11}^{10} & \cdot & K_{13}^{10} & \cdot & K_{15}^{10} & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & K_{11}^5 & \cdot & \cdot & \cdot & \cdot & K_{11}^{10} & K_{11}^{11} & \cdot & K_{13}^{11} & K_{14}^{11} & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_{12}^8 & K_{12}^9 & \cdot & \cdot & K_{12}^{12} & \cdot & \cdot & K_{15}^{12} & \cdot & K_{17}^{12} & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_{13}^{10} & K_{13}^{11} & \cdot & K_{13}^{13} & K_{14}^{13} & K_{15}^{13} & K_{16}^{13} & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_{14}^{11} & \cdot & K_{14}^{13} & K_{14}^{14} & \cdot & K_{16}^{14} & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_{15}^9 & K_{15}^{10} & \cdot & K_{15}^{12} & K_{15}^{13} & \cdot & K_{15}^{15} & K_{16}^{15} & K_{17}^{15} & K_{18}^{15} & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_{16}^{13} & K_{16}^{14} & K_{16}^{15} & K_{16}^{16} & \cdot & K_{18}^{16} & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_{17}^{12} & \cdot & \cdot & K_{17}^{15} & \cdot & K_{17}^{17} & K_{18}^{17} & K_{19}^{17} \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_{18}^{15} & K_{18}^{16} & K_{18}^{17} & K_{18}^{18} & K_{19}^{18} \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & K_{19}^{17} & K_{19}^{18} & K_{19}^{19}
\end{matrix}
$$

Figure 2.5: Stiffness matrix with a reduced profile corresponding to the variable numbering shown in figure 2.4.

2.5 Meshing

The meshing algorithm used in the original version of the program is very simple because most of the meshing work is in fact done by the user. The user writes code that generates a polygon file. This file format is illustrated in listing 2.2 and a graphical representation is shown in figure 2.6. The file describes a list of polygons, specifying the number of vertices, the type and the coordinates of the vertices.

The meshing algorithm takes this format as input and then proceeds in two phases. Firstly, a single central node is added to each polygon as illustrated in figure 2.7. Secondly, this mesh is optimised using the Delaunay algorithm which maximises the minimum angle of a triangle. The result is shown in figure 2.8. The data is stored in a field plot file of which the format is illustrated in listing 2.3. Three lists are stored in the file: a list of elements, a list of lines and a list of nodes. For an element, the numbers of its three nodes, its type and the numbers of its neighbouring elements are stored. For a line, simply the node numbers of its end points are stored. The list of nodes initially contains the coordinates of the nodes, and later, once the problem has been solved, the vector potential at the nodes is appended.

```
<number of polygons>
<number of points> <polygon type> <polygon number>
<point1 x> <point1 y>
<point2 x> <point2 y>
...
<number of points> <polygon type> <polygon number>
<point1 x> <point1 y>
...
```

Listing 2.2: Format of a polygon file.

```
3 (place holder for number of nodes per element)
<n elements> <n nodes> <vector potential flag> <n lines>
<element 1 node 1 node 2 node 3> <type> <neighbour 1 2 3>
<element 2 node 1 node 2 node 3> <type> <neighbour 1 2 3>
...
<line 1 node 1 node 2>
<line 2 node 1 node 2>
...
<node 1 x y a> (a is the vector potential at the node)
<node 2 x y a>
...
```

Listing 2.3: Format of a field plot file.

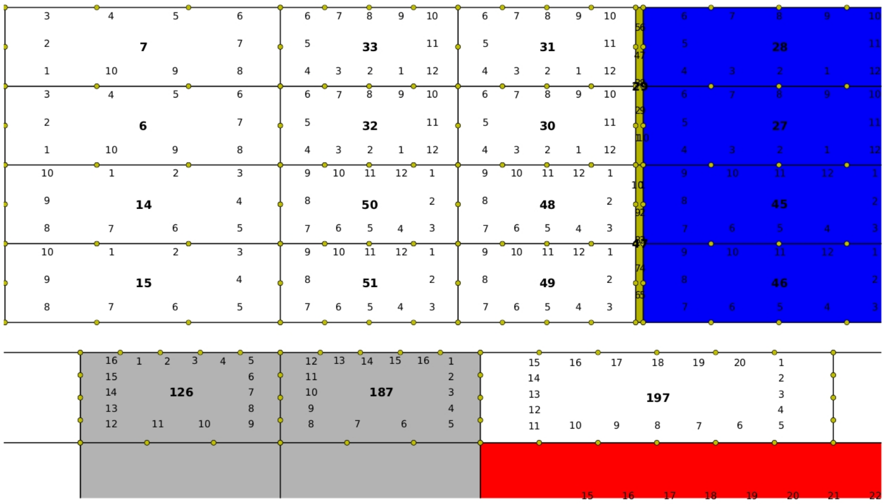


Figure 2.6: Graphical representation of a polygon file.

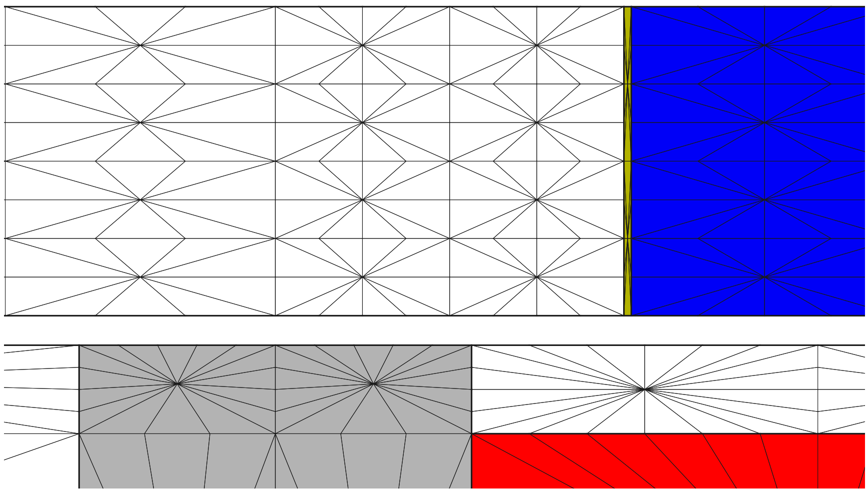


Figure 2.7: The mesh after the first phase of the meshing algorithm is complete.

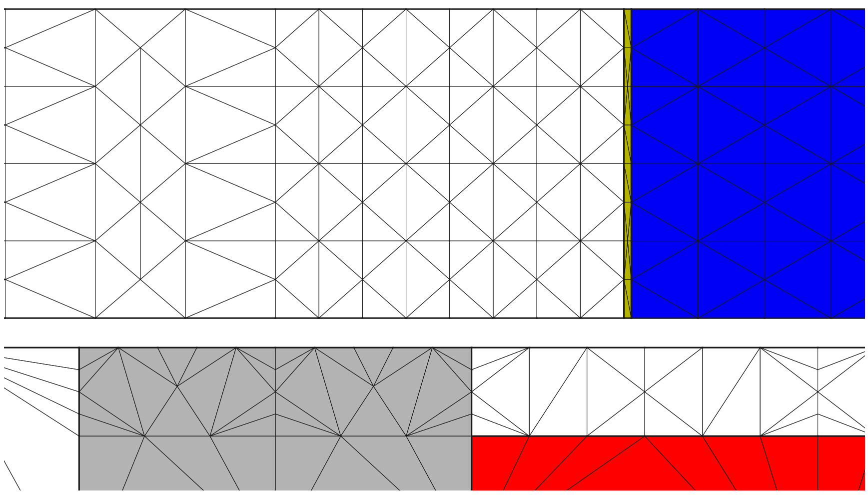


Figure 2.8: The mesh after Delaunay optimisation is complete.

2.6 Air-gap elements

2.6.1 Overview

The single most important distinction between this program and other packages used to simulate electrical machines in the present day is the way this program handles relative movement between different sections of the finite element mesh. The method used in this program, known as the air-gap element or macro element method was first proposed by Abdel-Razek et al. [2]. An air-gap element is a special kind of element used to connect two parts of a mesh that move relative to each other in a finite element simulation. A major advantage of using this method is that the forces acting on the moving component can easily be calculated using Maxwell's tensor, as is described in [1]. A linear version of the air-gap element method using the Cartesian coordinate system was derived by Wang [29, 30]. This air-gap element featured in the linear version of the program.

Originally, the air-gap element method was computationally very expensive, but Flack and Volschenk [8] developed a technique that greatly reduced the cost of the method. This technique was implemented in the original versions of the program and contributed to its good performance.

2.6.2 Basic theory

In this section, the air-gap element in its polar form, as derived by Abdel-Razek et. al. [2], is considered. The principles extend to the Cartesian form [29, 30] as well.

Fig. 2.9 shows the air-gap element between meshed sections of a machine. In the air-gap there is no current and the magnetic vector potential satisfies

$$\nabla \times (\nabla \times \mathbf{A}) = 0 \tag{2.7}$$

A solution to (2.7) can be obtained so that periodic boundary conditions on the dashed lines in Fig. 2.9 are satisfied and continuity with the solution of the traditionally meshed elements is maintained. This continuity is achieved by expressing both the solution and the boundary conditions as a Fourier series and equating coefficients. The solution takes the form

$$A_z(r,\theta) = \sum_{i=1}^{t} \alpha_i(r,\theta) u_i^{\varepsilon} \tag{2.8}$$

$$\alpha_i(r,\theta) = f_1(r)\frac{a_{0i}}{2} + \sum_{n=1}^{\infty} f_2(r)(a_{ni}cos(\lambda_n\theta) + b_{ni}sin(\lambda_n\theta)) \tag{2.9}$$

where t is the number of nodes connected to the air-gap element and the α_i fulfil the role of shape functions, similar to the N_i found in (5.6). The local stiffness matrix, $\mathbf{K}^{\varepsilon}$, is

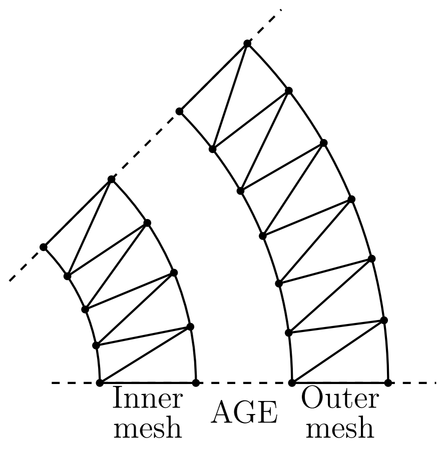


Figure 2.9: Air-gap element model.

constructed by minimising the energy functional as follows,

$$\frac{\partial E^{\varepsilon}}{\partial u_i^{\varepsilon}} = \frac{1}{\mu_0}\sum_{k=1}^{t} u_k^{\varepsilon}\int\int_{\Omega_{\varepsilon}} \nabla\alpha_i \cdot \nabla\alpha_k \mathrm{d}\Omega^{\varepsilon} = 0 \tag{2.10}$$

which leads to the matrix equation

$$\frac{\partial F^{\varepsilon}}{\partial \mathbf{u}^{\varepsilon}} = \frac{1}{\mu_0}\mathbf{K}^{\varepsilon}\mathbf{u}^{\varepsilon} = 0 \tag{2.11}$$

where $\mathbf{u}^{\varepsilon}$ is the vector of nodal values connected to the air-gap element.

Now the stiffness terms of nodes on the boundary of the air-gap element have contributions from their surrounding classical elements as well as the air-gap element. Thus, the general term of the global stiffness matrix is

$$K_{ij} = \sum K_{mn}^{e} + \sum K_{mn}^{\varepsilon} \tag{2.12}$$

where K^e represents the local stiffness terms of classical elements and K^{ε} represents the stiffness terms of the air-gap element.

An important difference between (1.3) and (2.8) is the number of contributing nodes. For first order triangular elements there are only three. This produces a sparse system matrix because a single node is typically not connected to many elements. Air-gap elements connect many nodes together and this causes the sparsity of the system matrix to be partially lost. This results in an increase in computational time needed to solve the system matrix equation, the main disadvantage of the air-gap element method.

2.7 2D solver

2.7.1 Overview

The original versions of the program could solve class I and class II problems. As mentioned in section 1.4.2, both these classes require the solution of (1.19).

In section 1.2.2 it was shown that applying the finite element method to solve (1.19) for the vector potential results in a system of linear equations

$$\mathbf{Ku} = \mathbf{f} \tag{2.13}$$

The matrix $\mathbf{K}$ is large (for example 10000×10000) and sparse. The size is very problem dependent and is a direct function of model size and mesh density. In any case, it is clear that an efficient method of solving the system equation (2.13) is required. Furthermore, because the coefficients of the matrix $\mathbf{K}$ are functions of the permeability μ according to (1.12) and μ is a non-linear function of $\mathbf{u}$, (2.13) should in fact be written as

$$\mathbf{K}(\mathbf{u})\mathbf{u} = \mathbf{f} \tag{2.14}$$

which can only be solved using some sort of iterative scheme that forces $\mathbf{u}$ to converge to the correct solution.

Considering the problem of solving (2.14), two algorithms are relevant. First, there is the algorithm used to solve a system of linear equations $\mathbf{Ku} = \mathbf{f}$ and secondly there is the algorithm used to update the solution vector $\mathbf{u}$ until convergence is achieved.

2.7.2 Algorithm used to solve a system of linear equations

The program solved the equation $\mathbf{Ku} = \mathbf{f}$ using the subroutine ACTCOL presented by Zienkiewicz [33]. This subroutine is based on the LU (lower-upper) decomposition of the matrix $\mathbf{K}$.

In the process of generating the stiffness matrix $\mathbf{K}$, the algorithm presented by Gibbs et al. [12] is used to improve the conditioning of the matrix. Although this is a costly operation, it is performed only once and reduces the time needed to solve the resulting equation sufficiently to easily justify its use. Note that an equation of the form $\mathbf{Ku} = \mathbf{f}$ must be solved multiple times in order for the solution of (2.14) to converge to the correct solution. Furthermore, in a time-stepped simulation (2.14) must also be solved for multiple positions of the rotor or translator. Thus, any improvement in the solution time of the equation $\mathbf{Ku} = \mathbf{f}$ will have a substantial impact on overall performance.

2.7.3 Algorithm used to update the solution vector

The Newton-Raphson method, which normally converges to the correct solution quickly[2], was used in the original versions of the program to update the solution vector $\mathbf{u}$. Using this method, the solution vector is updated at each iteration according to the following equation [3],

$$\mathbf{u}^{m+1} = \mathbf{u}^m - \mathbf{J}^{-1}[\mathbf{K}^m\mathbf{u}^m - \mathbf{f}] \tag{2.15}$$

where $\mathbf{J}$ is the Jacobian matrix of the vector

$$\mathbf{F} = \mathbf{K}^m\mathbf{u}^m - \mathbf{f} \tag{2.16}$$

with respect to the vector $\mathbf{u}$. This matrix is constructed by merging contributions from the local element systems just as for the stiffness matrix, as was explained in section 1.2.2. The local Jacobian matrix is then given by

$$\mathbf{J}^e = \begin{bmatrix} \dfrac{\partial F_1}{\partial u_1^e} & \dfrac{\partial F_1}{\partial u_2^e} & \dfrac{\partial F_1}{\partial u_3^e} \\[2ex] \dfrac{\partial F_2}{\partial u_1^e} & \dfrac{\partial F_2}{\partial u_2^e} & \dfrac{\partial F_2}{\partial u_3^e} \\[2ex] \dfrac{\partial F_1}{\partial u_3^e} & \dfrac{\partial F_3}{\partial u_2^e} & \dfrac{\partial F_3}{\partial u_3^e} \end{bmatrix} \tag{2.17}$$

The above leads to the general term of the local Jacobian matrix being given by

$$J_{ij}^e = K_{ij}^e + \frac{2}{A}\frac{\partial \kappa}{\partial (p^2)} R_i R_j \tag{2.18}$$

with κ the reluctivity, p the magnitude of the flux density and

$$R_m = \sum_{k=1}^{3} \mu K_{mk}^e u_k^e \tag{2.19}$$

In the above equations, small modifications were made from the notation found in Binns et. al. [3].

2.8 Torque and force calculations

The different versions of the program were capable of calculating the torque on the rotor and the force on the translator respectively. The program performed these calculations using the method based on the Maxwell stress tensor presented by Razek et al. [1]. The fundamentals of this method are considered shortly.

[2]The method may not converge if the initial estimate of the solution is not sufficiently close to the true solution. This problem can be solved by using another method to obtain a better initial estimate before switching to the Newton-Raphson method. (see section 3.6)

Using the Maxwell stress tensor, Stratton [26, p. 103] derived a formula for the total force acting on a body due to magnetic fields, namely

$$\mathbf{F}_m = \oint_\Gamma \left[\frac{1}{\mu_0}(\mathbf{B}\cdot\mathbf{n})\mathbf{B} - \frac{1}{2\mu_0}B^2\mathbf{n} \right] \mathrm{d}a \tag{2.20}$$

with Γ a surface enclosing the body, $\mathbf{n}$ a unit vector normal to Γ and $\mathbf{B}$ the magnetic flux density. The concept is illustrated in figure 2.10. Note that only a single side of the rectangular closed surface shown in figure 2.11 is considered in figure 2.10. This is because in all practical cases the contributions from the other three sides are zero. This may be due to periodic boundary conditions which eliminate two surfaces or the fact that either $B_x = 0$ or $B_y = 0$ on the other surfaces.

In the linear case shown, one is usually interested in the force in the direction of movement.

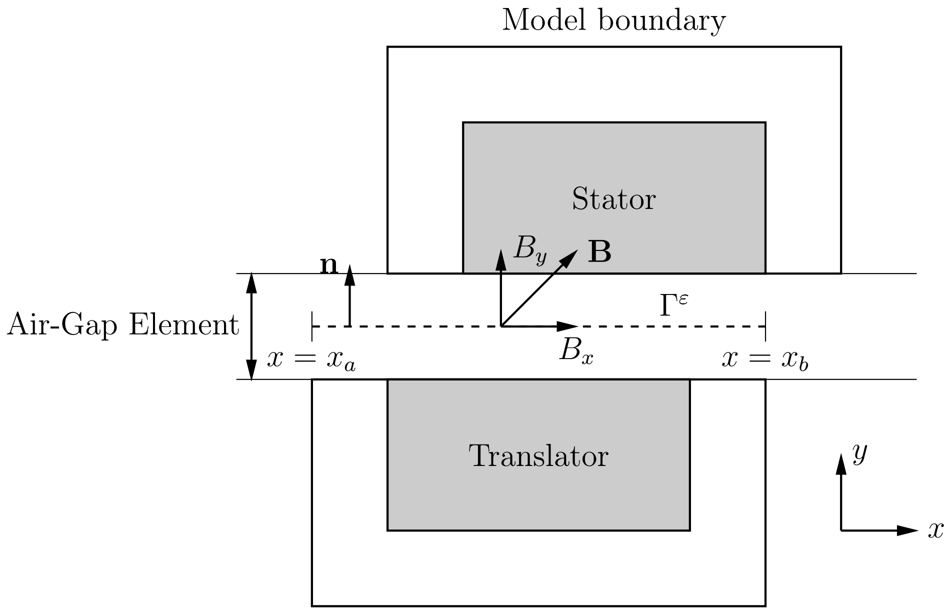


Figure 2.10: Force calculation for a linear machine.

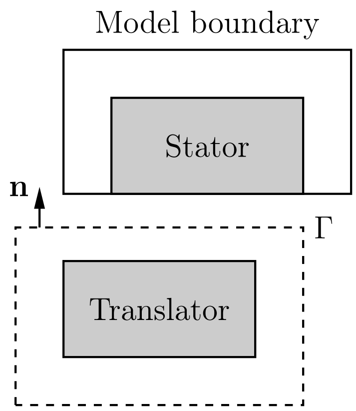


Figure 2.11: The entire closed surface integration path. Only the upper boundary in the air-gap contributes to the force on the translator.

This force is given by

$$F_x = \frac{w}{\mu_0} \int_{x_a}^{x_b} B_x B_y \mathrm{d}x \tag{2.21}$$

with w the model depth in the z direction. For the rotary case, the torque is given by

$$T = \frac{rw}{\mu_0} \int_{\theta_a}^{\theta_b} B_r B_\theta r \mathrm{d}\theta \tag{2.22}$$

One of the great advantages of using air-gap elements is the simplicity with which the above calculations can be carried out. Expressing the flux densities in terms of the vector potential ($\mathbf{B} = \nabla \times \mathbf{A}$) and substituting (2.8) into the above equations yield expressions for the force and torque

$$F_x = \frac{w}{\mu_0} (\mathbf{u}^\varepsilon)^T \mathbf{S}^{II} \mathbf{u}^\varepsilon \qquad T = -\frac{rw}{\mu_0} (\mathbf{u}^\varepsilon)^T \mathbf{S}^{I} \mathbf{u}^\varepsilon \tag{2.23}$$

with $\mathbf{u}^\varepsilon$ the vector of vector potentials of nodes on the boundary of the air-gap element and the matrix $\mathbf{S}$ is

$$S_{ij}^{II} = \int_{x_a}^{x_b} \frac{\partial \alpha_i}{\partial x} \frac{\partial \alpha_j}{\partial y} \mathrm{d}x \qquad S_{ij}^{I} = \int_{\theta_a}^{\theta_b} \frac{\partial \alpha_i}{\partial r} \frac{\partial \alpha_j}{\partial \theta} \mathrm{d}\theta \tag{2.24}$$

for class I and II problems respectively.

This method is capable of calculating torque or force very accurately because the path of the surface integral falls in a region where an analytical solution of the field is available.

2.9 Flux linkage calculation

The flux linkage of a coil is given by Haus et. al. [13] as

$$\lambda = \int_S \mu \mathbf{H} \cdot \mathrm{d}\mathbf{a} \tag{2.25}$$

where S is the surface enclosed by the coil and $\mathrm{d}\mathbf{a} = \mathbf{n} \cdot \mathrm{d}a$ with $\mathbf{n}$ the unit vector normal to the surface S with direction given by the right-hand rule. Using the magnetic vector potential, (2.25) may also be expressed as

$$\lambda = \int_S (\nabla \times \mathbf{A}) \cdot \mathrm{d}\mathbf{a} \tag{2.26}$$

Using Stokes' theorem, this can be written as

$$\lambda = \oint_C \mathbf{A} \cdot \mathrm{d}\mathbf{r} \tag{2.27}$$

with C the contour enclosing the surface S. The above equation is valid for a single turn coil where the cross sectional area of the wire is negligible. For a coil with N turns and non-negligible cross sectional area, the contour integral must be replaced by a volume integral and the above equation becomes

$$\lambda = \frac{N}{V_C} \oint_{V_C} (\mathbf{A} \cdot \mathbf{n_C}) \mathrm{d}V \tag{2.28}$$

with V_C the volume occupied by the coil and $\mathbf{n_C}$ a unit vector in the direction of positive current flow as defined by the right-hand rule. For the class I and II problems considered in the original program, $\mathbf{A} = A_z$ and the volume integral can be replaced by a surface integral to yield

$$\lambda = wN \left[\int_{S_{C+}} \frac{A_z}{S_{C+}} \mathrm{d}a - \int_{S_{C-}} \frac{A_z}{S_{C-}} \mathrm{d}a \right] \tag{2.29}$$

with w the model depth in the z direction. In the above equation the surface integral has been split into two parts. The surface S_{C+} is the area where the positive directions of A_z and $\mathbf{n_C}$ coincide. On S_{C-} these directions are opposites.

The program implemented (2.29) in a single subroutine that could be called to calculate the flux linkages of coils once a solution of the vector potential A_z was available. The continuous surface integrals being replaced by a summation of the average vector potential on all coil elements, to obtain

$$\lambda = \frac{wN}{A_C} \left[\sum \frac{\gamma A \sum_{i=1}^{3} u_i}{3} \right] \tag{2.30}$$

with A the area of an element, A_C the total cross sectional area of the coil and $\gamma = 1$ or $\gamma = -1$, depending on whether the element is part of S_{C+} or S_{C-}.

2.10 Conclusive remarks

In this chapter, the original program received by the author was discussed in some detail. Several important concepts and calculations have been documented. With the foundation that has been laid in this chapter, the improvements and extensions that were made to this program will be discussed in the following chapters.

Chapter 3

Improvements

3.1 Introduction

Having described the original program in some detail, this chapter goes on to discuss some of the improvements and extensions that were made during the course of this work. It is shown how the issues mentioned in section 2.3 were addressed and how new functional capabilities were implemented in the program. The improved program, which is hardly recognisable as a modification of the original, has been dubbed *SEMFEM*: the Stellenbosch Electrical Machines Finite Element Method.

The chapter begins by taking a look at the restructuring process and the creation of the *SEMFEM* core in section 3.2. After a short note on compilation in section 3.3, two alternative methods of generating meshes that were implemented in *SEMFEM* are discussed in section 3.4. In section 3.5, magnet modelling is considered, presenting a more generic method which required only small modifications to (1.12). A few additions to the general non-linear solver which improved computational efficiency and robustness are discussed in section 3.6. A very useful collection of graphical output tools are presented in section 3.7. Finally, in section 3.8, some attention is given to several calculations that can be placed in the post-processing category.

3.2 Restructuring

The aim of the restructuring process was primarily to address the issues mentioned in section 2.3. This part of the work was not concerned with the mathematics of the program but rather the form of the code. The aim was to make it easier to understand and work with the program, while maintaining its functionality and good performance.

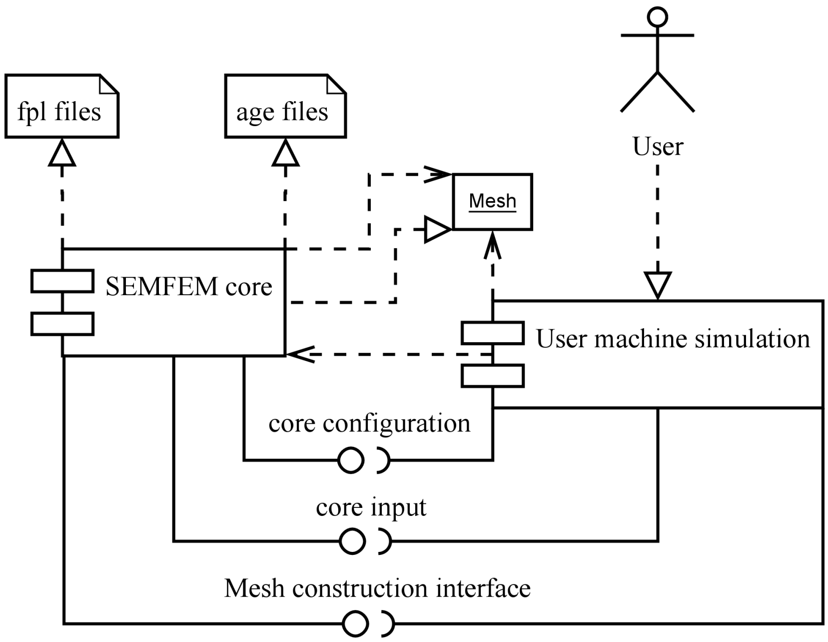


Figure 3.1: The program after the restructuring process.

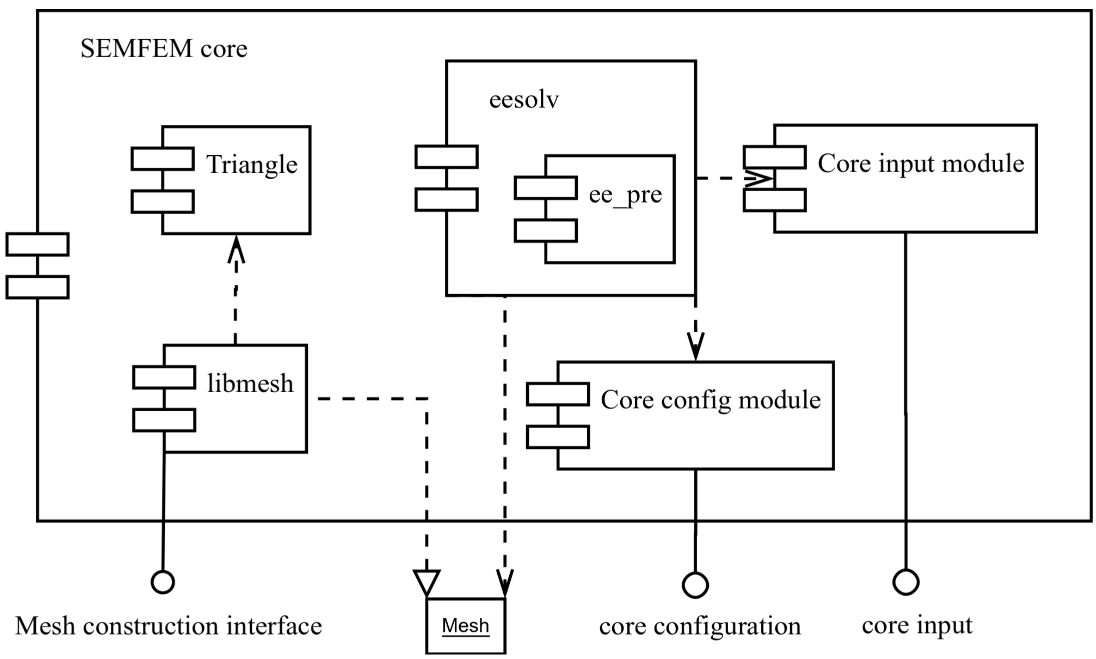


Figure 3.2: The inner structure of the core.

The improved structure is shown in figures 3.1 and 3.2. All the functionality required to generate a mesh, simulate a machine and calculate forces and flux linkages is built into the *SEMFEM* core component. As can be seen in figure 3.1, the mesh – which has been declared as a derived type – is exposed to the user program. This allows the user to access all the information encapsulated in the mesh structure directly. The core implements functionality to generate the mesh, but if the user wishes to generate the mesh in some other way, the core is perfectly capable of using a mesh generated using another method, provided that is in the correct format. The mesh structure is discussed in more detail later in this section. Note also that the core is still capable of generating field plot files and air-gap element files, but these files are no longer necessary for the program to function. The core can also generate other output files not shown in the figures containing detailed information on movement, currents, flux linkages and forces. Figure 3.1 also illustrates the separation between the *SEMFEM* core and the user simulation. All the user's work is done in the *User machine simulation* component. Apart from the mesh, information required by the core must be provided by the user through the core configuration and input interfaces. These interfaces will be discussed in more detail later in this section.

Figure 3.2 shows the internal structure of the core. The *Triangle* and *libmesh* components are responsible for mesh generation and are discussed in section 3.4.2. The function of the *Core config module* and *Core input module* components is to store information provided by the user about the simulation to be run. These components also implement the functions that are used by the user to initialize the required information. The *eesolv* component is responsible for solving the system equation at each time-step of the simulation and calculating flux linkages and forces.

The most important changes that were made during the restructuring process can be summarised as follows:

- The original program was written for the GNU f77 compiler. It was decided that the GNU gfortran compiler would be used instead. Currently, this compiler conforms to the Fortran 95 standard and also implements some features of the Fortran 2003 standard. The main advantages of using the newer compiler is that it allows the use of free-form source code (which is much more pleasant to work with) and powerful new program units called modules.
- All file operations between the different components of the package were removed. All data is stored in memory.
- A clear distinction was made between the core program and machine specific code. This simultaneously removed the problem of a complicated input mechanism and the need to keep a different version of the main components for every different machine.

In this way, the *SEMFEM* core can be distributed as a single shared object (.so) or a dynamic link library (.dll) that contains most of the complex functions required for finite element analysis. A user can then use *SEMFEM* by simply linking the user simulation code against the *SEMFEM* library.

- All arrays used in the program were originally declared to have a fixed size. For smaller problems this results in inefficient use of memory. This also places a limit on the size of problem that can be solved using the program. These issues can be overcome by using dynamic memory allocation. Several of the large arrays used in the program were either converted to dynamic arrays or modified so that their size can be modified easily using a global parameter. The *MESH_SIZE* parameter in listing 3.1, which will be discussed in the next point, is an example of the second approach.

- Derived types were created to store polygon, mesh and air-gap element structures. This allows these whole structures to be passed to a subroutine as a single argument. This strategy avoids both excessively long function interfaces and excessive use of global variables. The declarations of the mesh structure is shown in listing 3.1. It contains all the data originally stored in the field plot file as well as some other variables that will not be discussed here.

Comparing figures 3.1 and 3.2 with figure 2.1, it is evident that the structure is simplified from the user's perspective and that the core essentially behaves as a black box meaning

```
type :: mesh
    integer nde
    integer nelm, nnode, nlines
    integer node(MESH_SIZE, 3)
    integer node_type(MESH_SIZE)
    integer itype(MESH_SIZE)
    integer neigh(MESH_SIZE, 3)
    integer mark(MESH_SIZE)
    integer line(MESH_SIZE, 2)
    double precision, dimension(MESH_SIZE) :: x, y, a
    double precision, dimension(MESH_SIZE) :: ar, b
    ! coil areas
    double precision acoil(NCOIL), scoil(NCOIL)
    ! coil resistance
    double precision rcoil(NCOIL)
    ! meshed air-gap
    integer old_nelm
    double precision current_pos
end type
```

Listing 3.1: The derived type used to store all information related to the mesh.

that the user does not have be concerned with that part of the implementation.

The core needs to provide more functionality than any of the original versions of the program because the same core is used for different problem classes and different configurations. It does this by using configuration parameters that can be set by the user through a call to a configuration initialization subroutine. This interface is the core configuration interface shown in figures 3.1 and 3.2. The most significant configuration parameters are the following:

- **Problem type:** This parameter indicates to what class the problem belongs. The core uses this parameter to select the appropriate routines for the specific problem class. The calculation of the local stiffness matrices, for example, is the same for class I and II problems, but differs for class III problems.
- **Air gap method:** This parameter controls whether there are air-gaps in the model that should be modelled using the air-gap element technique and if so, how many.

Other parameters include a file name and parameters that control whether or not field plot or air-gap element files are generated.

Apart from the configuration parameters, there are also some other variables in the core that need to be initialized correctly by the user. This is done through the core input interface. These variables are the following:

The number of time steps This parameter controls how many solutions are required. The user is free to vary currents and/or positions from time-step to time-step as is required.

Rotor/translator positions at each time step This array is used to set the relative shift between air-gap element boundaries at each time-step. In this way, the position of different parts of the mesh can be controlled at each time-step.

Winding currents at each time step This array defines the instantaneous value of the currents flowing in the windings at each time-step.

Winding turns This array defines how many turns each winding has.

Winding scale factors This array is used to facilitate the situation where a single winding is formed by connecting coils in series. It is specifically used to calculate the current density and flux linkages. For example, if a winding consists of a single coil, the current density is given by the total current divided by the cross-sectional area of the coil. If the winding, however, consists of two coils, the program will incorrectly calculate the cross-sectional area of the winding as the sum of the cross-sectional areas of the two coils. In this case, the winding scale factor should be set to two in order to rectify the problem.

Model depth This parameter is applicable to class I and II problems and sets the depth of the model. This depth is used to scale the calculated flux linkages and forces. For class III problems, it is assumed that the model is for a complete rotation about the z-axis and thus, the flux linkages and forces are scaled using a model depth of $2\pi r$.

Air-gap positions These arrays contain the coordinates (y-coordinate for class I and II, r-coordinate for class III) of the upper and lower edges of the air-gap elements.

Boundary conditions This array is used if the boundary conditions are not explicitly specified with the functions discussed in section 3.4.2. It specifies the boundary conditions to be applied on four edges of a model. For class I, the model is assumed to be shape like an arc segment. For class II and III, the model is assumed to be rectangularly shaped. The core then attempts to auto-detect the boundary edges and mark the nodes according to the specified boundary conditions.

In the opinion of the author, the restructuring process greatly improved the usability and maintainability of the program. The addition of the capabilities to handle multiple air-gaps and solve class III problems, discussed in chapters 4 and 5 respectively, would have been much more difficult if the restructuring process was not completed first.

3.3 A note on compilation

The original program received by the author came with a Makefile that could be used to compile the source code and link the object files to create the executable program. One of the things to consider when compiling code is the level of optimisation used. No optimisation results in fast compilation. Higher levels of optimisation will result in longer compilation time but improved performance. The original Makefile received by the author used the `-O` flag, specifying the first level of optimisation. It was found that using the `-O3` flag, the maximum level of optimisation, improved the performance of the program significantly. The results from three test cases illustrating this improvement are shown in table 3.1. On average, the use of the `-O3` flag resulted in about a 50% improvement in computational time! In the case of this project, compilation times were short, typically around 7 seconds for the entire core using the `-O3` flag and about half the time using the `-O` flag. Thus, the increase in compilation time was totally insignificant, bearing in mind that the core does not need to be recompiled every time the user's program changes.

Table 3.1: Benchmarking of different optimisation levels.

Test case	`-O` [seconds]	`-O3` [seconds]
1	1324	455
2	19	11
3	127	67

Optimisation for specific processors is also possible using the `-mtune=cpu-type` option. This was not used during the course of this work, but it is recommended that the benefits of this option is investigated in the future.

3.4 Mesh generation

As mentioned in section 2.3, a more convenient method of constructing the finite element mesh was required. During the course of this work, two new methods of constructing the mesh were developed. Both of these methods greatly simplify the process of drawing a machine in a format compatible with the simulation program. The first method relies on creating customizable slot classes, using the open source scripting language *Python*, which could be assembled into a complete machine model. The creation of these slot classes is simplified by a library of convenience functions. The second method incorporates the open source mesh generator *Triangle* [22] into the program. Here follows a more in depth discussion on both these methods.

3.4.1 Mesh generation using Python slot classes

The first attempt at improving the mesh generation process was based on the idea of constructing a mesh from a number of predefined slots. These slots were realised by dedicated Python classes. The reasons for implementing this method in Python are as follows:

- The method was well suited to an object oriented approach.
- Many potential users of the program were proficient in Python. At the time, it was considered to create a fully functional Python binding for *SEMFEM*.[1] In that case, there was no reason to use another language for mesh generation.

[1] The possibility of creating a Python binding for *SEMFEM* still exists. The way *SEMFEM* was restructured with well defined interfaces between the *SEMFEM* core and the user code may simplify the implementation of a Python binding greatly.

```
#!/usr/bin/python
from __future__ import division

import sys
sys.path.append("./")

from polpy.polpy import *
from polpy.rfapm_rotorslot import rfapm_rotorslot as rslot
from polpy.rfapm_statorslot import rfapm_statorslot as sslot

import post2

def assemble(args):
    ro_id = args[0]
    ro_od = args[1]
    ro_mh = args[2]
    ro_ma = args[3]
    ro_ih = args[4]
    n_poles = args[5]
    sheet = args[6]
    st_id = args[7]
    st_od = args[8]
    st_yh = args[9]
    st_cw = args[10]
    st_thb = args[11]
    n_slots = args[12]
    st_md = (st_id + st_od)/2.0
    n_slots = 24

    rs1 = rslot(st_md,ro_id,ro_od,ro_mh,ro_ma,ro_ih,n_poles,sheet)
    rs2 = rslot(st_md,ro_id,ro_od,ro_mh,ro_ma,ro_ih,n_poles,sheet)
    rs3 = rslot(st_md,ro_id,ro_od,ro_mh,ro_ma,ro_ih,n_poles,sheet)
    rs4 = rslot(st_md,ro_id,ro_od,ro_mh,ro_ma,ro_ih,n_poles,sheet)
    slots = [rs1, rs2, rs3, rs4]
    rot = rotor()
    rot.assemble(slots)

    ss1 = sslot(st_id, st_od, st_cw, n_slots, -1, 3)
    ss2 = sslot(st_id, st_od, st_cw, n_slots, -3, 2)
    ss3 = sslot(st_id, st_od, st_cw, n_slots, -2, 1)
    slots = [ss1, ss2, ss3]
    stat = stator()
    stat.assemble(slots)

    mac = machine()
    mac.assemble(rot,stat)

    return mac.pol();
```

Listing 3.2: Example of a Python mesh generating script.

- The computational time needed to generate a mesh using this method is negligible compared to the rest of the simulation. Thus, the slower performance of Python is not an issue.
- Many things are simply easier to implement in Python.

The idea was to have a library of these slot files that could be easily assembled into a complete machine. If a user wanted to simulate a machine that could not be described by existing slot classes, all that was needed was to create a new slot class. The process of assembling different slots into a complete machine was largely automated by a library.

The implementation of this method was rather tricky because it involved embedding a Python interpreter in a C function that could be called from Fortran. In this way data can be passed from a Fortran subroutine to a Python method. Apart from the *user machine simulation* component shown in figure 3.1, an accompanying Python script that generates the mesh structure is needed for this method. An example of such a script is shown in listing 3.2. In the listing, the dimensions of the machine are passed to the assemble function in the array *args*. Four identical rotor slots are assembled to form the rotor. Three stator slots with different windings are assembled to form the stator. The rotor and stator are then assembled into the complete machine model. All mesh generating scripts using this method are as simple as the own shown. The catch is that if an appropriate slot class does not exist, a user must implement it. Depending on the geometry and the number of variables for the slot, the implementation of these slot classes may require a significant amount of work.

While this method was an improvement over the original method of mesh generation, it was still not satisfactory. The primary reason for this being that there was no way of controlling the mesh density easily. Changing the mesh density required modification of the slot classes. Although this was easier than the original method, it was still a cumbersome exercise. The fact that the mesh density could not be controlled easily was considered a significant drawback as far as the usability of the program was concerned.

3.4.2 Mesh generation using Triangle

This method is the most automated and requires the least amount of effort from the user. It is also the most powerful because it allows great flexibility in setting the mesh density in different regions or along specific lines. Although *Triangle* does have a simple input format, the process of drawing a machine was further simplified by developing a library that aids the user in generating the input required by *Triangle*. This library provides the mesh generation interface shown in figures 3.1 and 3.2.

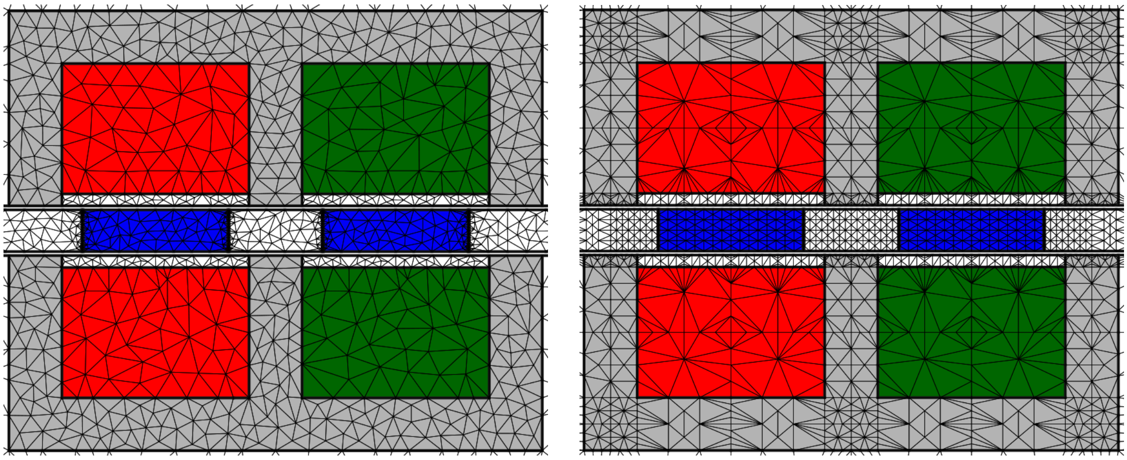


Figure 3.3: Meshes generated with *Triangle* and Python respectively.

To illustrate the use of this method, the user code that is required to generate the meshes shown in figure 3.4 is given in listing 3.3. All the subroutines called in this code snippet are from the library component, *libmesh*, shown in figure 3.2. This library, like *Triangle* itself, is implemented in C. It is approximately 900 lines long and contains all the functionality to convert data structures from the representation used by *Triangle* to the representation used by *SEMFEM* as well as the subroutines that users of the program need to construct a mesh. Comparing listings 3.2 and 3.3, it may appear that the method using *Triangle* will require more work from the user compared to the method using Python slot classes, but this is not the case. Firstly, if an appropriate Python slot class does not exist, the user has to create it. Creating these slot classes are typically much more demanding than creating the assemble script in listing 3.2. Secondly, the *Triangle* method allows easy manipulation of the mesh density in various parts of the model through the *region* subroutine. For example, the only difference in the code required to generate the meshes in figure 3.4 is that the `mesh_scale` parameter was changed from 1 to 4. This is a vast improvement over the method using Python slot classes which requires modification of the slot classes.

The drawing subroutines available to the user are documented here. Note that the C function signatures are shown. The underscores are appended to the names of the functions because the Fortran compiler appends an underscore to external symbols. Thus, a C function like void `init_draw_()` can be called from Fortran using `call init_draw()`.

void init_draw_()

This function initializes a global mesh structure to which subsequent calls to drawing functions add data.

void region_(double *x, double *y, double *type, double *maxarea)

This function assigns a material type and a mesh density to a bounded region of the mesh containing the point (x, y).

```
subroutine draw(m)
    type(mesh) :: m
    double precision :: x1, x2, y1, y2, sheet
    double precision :: mesh_scale = 1d0
    double precision :: air_mesh, yoke_mesh, mag_mesh, seg_len

    air_mesh = mesh_scale*90d-6
    yoke_mesh = mesh_scale*10d-6
    mag_mesh = mesh_scale*30d-6
    seg_len = mesh_scale**0.5 * 2d-3
    sheet = mag_width/100

    call init_draw()

    call oarc(0d0, 2*pi, r1, seg_len)
    call oarc(0d0, 2*pi, r2, seg_len)
    call boarc(0d0, 2*pi, r3, seg_len, DIRICHLET)
    call hole(0d0, (r1+r2)/2)
    call region(0d0, r1 - 1d-3, 0d0, air_mesh)
    call region(0d0, r2 + 1d-3, 250d0, yoke_mesh)

    ! Magnet
    x1 = -mag_width/2
    x2 = mag_width/2
    y1 = -mag_height/2
    y2 = mag_height/2
    call line(x1, y1, x1, y2)
    call line(x1, y2, x2, y2)
    call line(x2, y2, x2, y1)
    call line(x2, y1, x1, y1)
    call region(0d0, 0d0, 12d0, mag_mesh)

    ! Current sheets
    x1 = -mag_width/2 - sheet
    x2 = -mag_width/2
    call line(x2, y1, x1, y1)
    call line(x1, y1, x1, y2)
    call line(x1, y2, x2, y2)
    call region((x1+x2)/2, 0d0, 4d0, mag_mesh)
    x1 = mag_width/2 + sheet
    x2 = mag_width/2
    call line(x2, y1, x1, y1)
    call line(x1, y1, x1, y2)
    call line(x1, y2, x2, y2)
    call region((x1+x2)/2, 0d0, -4d0, mag_mesh)

    call make_mesh(m)
end subroutine
```

Listing 3.3: Example of mesh generation using *Triangle* and the mesh generation interface provided by the *SEMFEM*_core component.

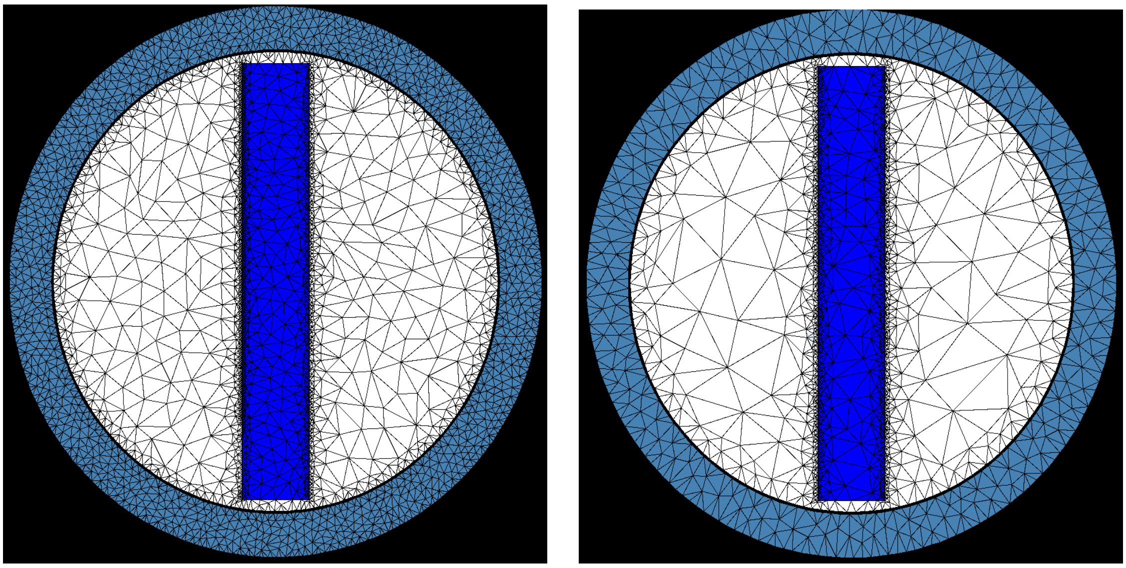


Figure 3.4: Meshes generated with *Triangle*.

void pregion_(double *r, double *theta, double *type, double *maxarea)

Same as the above, except that the coordinates of the points is expressed in polar form.

void oarc_(double *theta1, double *theta2, double *r, double *max_len)

This functions draws an arc with radius r in a counter-clockwise direction starting at theta1 and ending at theta2. The arc is constructed from straight line segments so that no segment is longer than max_len.

void boarc_(double *theta1, double *theta2, double *r, double *max_len, int *btype)

Same as oarc_ but the generated nodes are also marked as boundary nodes with type btype. The possible values for btype are DIRICHLET, NEUMANN, EVEN_PERIODIC or ODD_PERIODIC.

void line_(double *x1, double *y1, double *x2, double *y2)

This function draws a single segment line from (x_1, y_1) to (x_2, y_2).

void pline_(double *r1, double *theta1, double *r2, double *theta2)

Same as the above, except that the coordinates of the points are expressed in polar form.

void nline_(double *x1, double *y1, double *x2, double *y2, int *nseg)

Same as line_ except that a line of nseg equal segments is constructed.

void pnline_(double *r1, double *theta1, double *r2, double *theta2, int *nseg)

Same as the above, except that the coordinates of the points are expressed in polar form.

void mline_(double *x1, double *y1, double *x2, double *y2, double *max_len)

Same as line_ except that an equally segmented line is constructed so that the segments are not longer than max_len.

void pmline_(double *r1, double *theta1, double *r2, double *theta2, double *max_len)

Same as mline_ except that the coordinates of the points are expressed in polar form.

void bmline_(double *x1, double *y1, double *x2, double *y2, double *max_len, int *btype)

Same as mline_ but the generated nodes are also marked as boundary nodes with type btype. The possible values for btype are DIRICHLET, NEUMANN, EVEN_PERIODIC or ODD_PERIODIC.

void bnline_(double *x1, double *y1, double *x2, double *y2, int *nseg, int *btype)

Same as nline_ but the generated nodes are also marked as boundary nodes with type btype. The possible values for btype are DIRICHLET, NEUMANN, EVEN_PERIODIC or ODD_PERIODIC.

void mirror_(double *ax_pos, char *ax)

This function adds a copy of all lines and nodes mirrored about the specified axis. The parameter *ax* can be either "x" or "y" and ax_pos specifies the distance of the axis from the origin.

void hole_(double *x, double *y)

This function marks a bounded region containing the point (x, y) as a hole in the mesh.

void make_mesh_(mesh *m)

This function performs the final preparation of the input data, calls *Triangle* to create the mesh and copies the generated data to the mesh structure, m, used by *SEMFEM*.

3.5 Magnet modelling

The original version of the program modelled magnetic materials with equivalent current sheets. Examples of magnets modelled using this method are shown in figure 3.5. The light yellow on the left sides of the magnets represents a current sheet with a current density directed into the page and the darker yellow on the right sides, a current sheet with current directed out of the page. In the blue area there is no current and the permeability is the magnet's recoil permeability. The magnitude of the current density is calculated from the characteristics of the magnet. To model magnets such as the ones

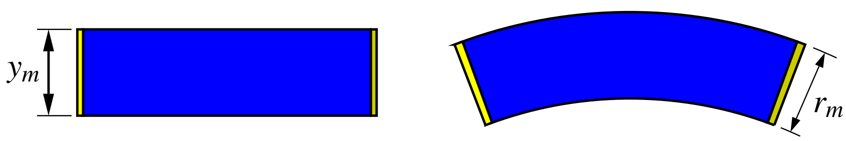


Figure 3.5: Magnets modelled using current sheets.

shown in figure 3.5, the total current in the sheets is given by

$$I_s = y_m H_c \quad \text{and} \quad I_s = r_m H_c \tag{3.1}$$

for the two magnets respectively, with H_c the coercive force. This method has the disadvantage of making the model more complex since every magnet requires two current sheets. From the user's perspective, this makes the model more complex to draw. It also limits the type of magnets that can be modelled because magnets with more complex geometries or magnetisation directions are not easily modelled using this technique.

A better alternative is to account for the magnetisation using an additional source term in Ampère's law as given by Binns et. al. [3], namely

$$\nabla \times \frac{1}{\mu}(\nabla \times \mathbf{A}) + \nabla \times \frac{\mathbf{M}_0}{\mu} = \mathbf{J} \tag{3.2}$$

In the above equation, μ is the recoil permeability and $\mathbf{M}_0$ is the vector of the remanent flux of the permanent magnet material. Following the same procedure as in section 1.2.2, but using equation (3.2) instead of (1.1) leads to the coefficients of the system equation being given by

$$K_{ij}^e = \int_{\Omega^e} \frac{1}{\mu} \nabla N_i \nabla N_j \mathrm{d}\Omega^e \qquad f_i^e = \int_{\Omega^e} N_i J_z \mathrm{d}\Omega^e + \int_{\Omega^e} \frac{1}{\mu} \mathbf{M}_0 \nabla N_i \mathrm{d}\Omega^e \tag{3.3}$$

Notice that the equation for K_{ij}^e is unchanged, but there is an extra term in the equation for f_i^e.

3.6 General solver

A couple of improvements were also made to the non-linear solver in general. Originally the initial estimate of the vector potential was set to zero at every node for every time-step. It was found that using the solution of the previous time-step as an initial estimate for the current time-step resulted in quicker convergence to the solution. In other words, the solution of the previous time-step is a better initial estimate than the all zero approach used originally. In the one case where this improvement was measured, the simulation using the previous solution as an initial estimate was about 15% faster.

The original non-linear solver used default values for the reluctivity at every element as an initial estimate and then applied the Newton-Raphson method to find the correct solution. In some cases the author experienced that this method did not converge to the correct solution. This problem is also mentioned by Binns et al. [3, p. 296] where it is suggested that a simple iterative method be used to obtain a good initial estimate of the solution. At this stage a simple iterative solver had already been implemented by the author to test the implementation of the axisymmetric solver prior to the calculation of the axisymmetric Jacobian matrix. It was found that using this simple iterative solver to obtain an initial estimate and then switching to the Newton-Raphson method did result in more consistent convergence. Unfortunately, a performance penalty is incurred using this method. Consistent convergence is, however, very important when the program is used for optimisation.

Now, there are many ways to handle this trade-off between performance and consistent convergence. When the program is used to evaluate a single design, it is probably best to disable the simple iterations unless problems occur. Also when performing optimisation, the simple iterations may be disabled if performance is critical and the topology is not prone to problems with convergence. Another solution would be to attempt to solve without the simple iterations each time and only to use them in case convergence is not achieved. This would be a good choice unless starting with the Newton-Raphson method fails to achieve convergence regularly, in which case it would be best to always start with the simple iterations.

3.7 Graphical output capabilities

In this section all issues related to graphical output are discussed. There are two areas where graphical output is important. Firstly, it is important during the drawing process. A user should be able to inspect the generated mesh to ensure it is fine enough in all areas of the model. Boundary conditions can also be inspected to verify the correctness of a model. Secondly, after a simulation is completed the user may want to view a plot of the magnetic fields, view detailed flux densities or vector potentials and plot voltages, currents, etc.

All vector potentials and flux densities can be stored in the field plot files discussed in section 2.5. The field plot file format was slightly modified to store some additional data. A Python application was developed that can interpret these files and display the content graphically. The application is capable of displaying element numbers, node numbers, variable numbers, elemental flux densities, nodal vector potentials, flux lines, the mesh, colour coded material types and a colour map of the flux density. A couple of screen shots of the application is shown in appendix F.

A script also exists to display the contents of a .pol file, although .pol files are no longer required if *Triangle* is used for mesh generation.

Triangle also came with its own application for viewing meshes. Some work was necessary in order to use this program for displaying material types in different colours, but in the end this was accomplished. The necessary functionality was incorporated into the *libmesh* component. The advantage of the viewer from *Triangle* is that it is much faster than the developed Python application which is based on *matplotlib* [17], a 2D plotting library. The Python application is richer in functionality, however. Unfortunately, the viewer from *Triangle* only works on Unix, while the Python application is fully cross-platform.

Figure 3.3 was produced using the Python application while figure 3.4 was produced using the viewer from *Triangle*. Flux density colour maps such as the one shown in figure 7.17 can only be produced using the Python application.

3.8 Post-processing

Although the program is currently designed so that most post-processing calculations are made in the user-space and can be easily adapted for every machine, some templates have been designed with post-processing calculations for specific applications.

3.8.1 Short stroke linear machines

A particularly interesting case is simulations of short stroke linear machines such as the case studies presented in sections 7.3, 7.4 and 7.6. An important difference between this type of simulation and that of rotary machines is that the speed of the rotor is constant in the latter case but the speed of the translator varies in the former case. This implies that either the displacement of the translator or the time-steps have to vary at each step in time-stepped simulations of short stroke linear machines. For the case of sinusoidal movement, it was decided to implement a constant displacement of the translator at every time-step and to vary the time-steps sinusoidally. In this way, accurate flux linkage and force data is obtained over the entire range of translator displacements. However, when this data is plotted as a function of time, the data points are not evenly spaced, but closer together where the translator speed is high and further apart where the translator speed is low. For a half period simulation, the time at time-step n out of N is then given by

$$t(n) = \frac{1}{2\pi f} \arccos(1 - \frac{2n}{N}) \tag{3.4}$$

where f is the frequency of oscillation.

3.8.2 EMF calculation

The EMF developed in a winding is given by the time-derivative of the flux linking the winding (Haus et. al. [13]),

$$emf = \frac{\mathrm{d}\lambda}{\mathrm{d}t} \tag{3.5}$$

The original program evaluated this derivative using backward differences, i.e.

$$emf(t = t_i) = \frac{\lambda_i - \lambda_{i-1}}{t_i - t_{i-1}} \tag{3.6}$$

It was found that the derivative can be calculated more accurately if cubic splines are used to represent the flux linkage function and the analytic derivative of the cubic spline is used. The improvement in accuracy is most notable when data points are spaced further apart in time. As mentioned in the previous subsection, the simulation of short stroke linear machines produces points which are closely spaced in certain areas, but further apart in others. The use of splines allows accurate calculation of the derivative in the entire simulation period without using an excessive amount of data points.

The superior accuracy of the spline derivative is illustrated in figures 3.6 and 3.7 where different numerical differentiation techniques are compared. The figures show the voltage waveforms for a short stroke linear machine.

3.8.3 Conservation of energy

In this section, a simple necessary condition for the accuracy of some of the data used in post-processing will be discussed. It can also, in a sense, be used to get a measure of the practical accuracy of a simulation.

In a time-stepped simulation, the speed of the moving component (rotor or translator) is a given, while the force or torque on this component is calculated using the simulation. With these two quantities known, the instantaneous mechanical input power to the simulated machine can be calculated over the simulated time interval, e.g. for a linear machine the instantaneous power is given by

$$p_m(t) = f_m(t)v(t) \tag{3.7}$$

with $f_m(t)$ the force on the translator and $v(t)$ the speed of the translator. Integrating the instantaneous power, one obtains the total mechanical input energy to the system.

$$E_m = \int_0^T p_m(t)\mathrm{d}t \tag{3.8}$$

Referring to figure 3.8, the instantaneous electrical input power of the machine can be obtained from

$$p_e(t) = v_\lambda(t)i(t) \tag{3.9}$$

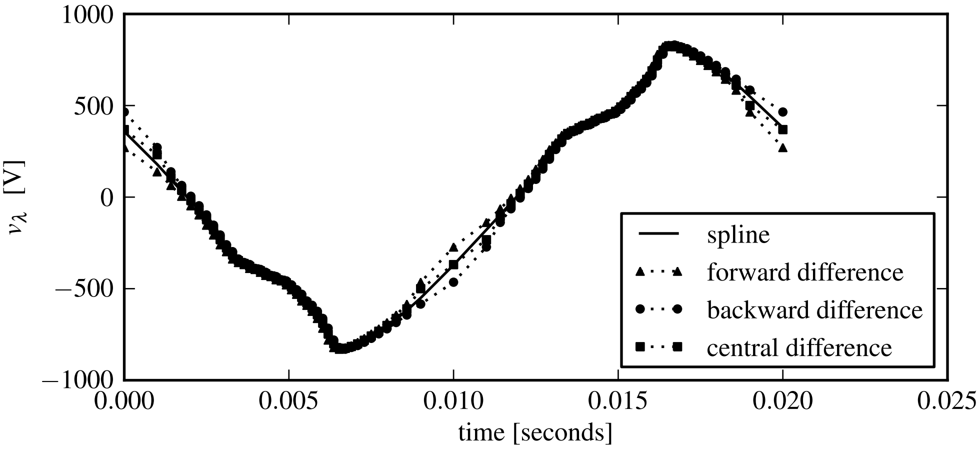


Figure 3.6: Comparison of differentiation techniques.

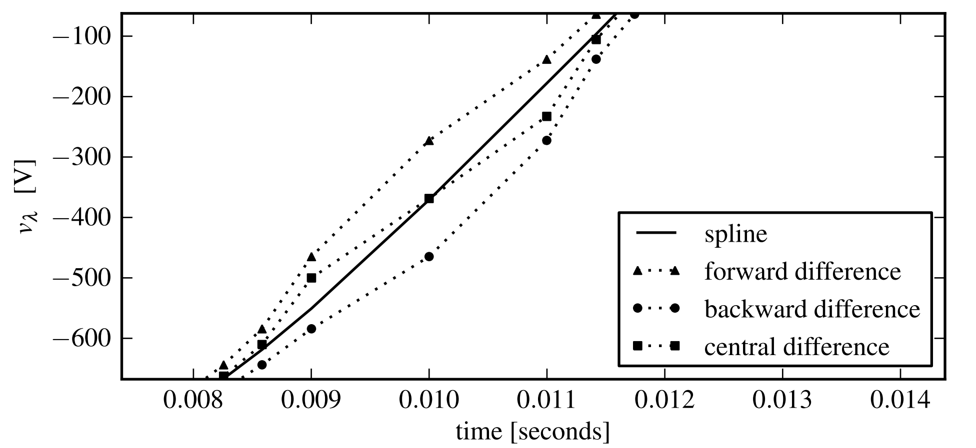


Figure 3.7: Comparison of differentiation techniques (close up).

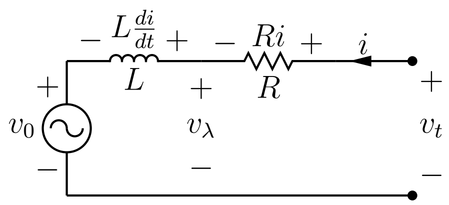


Figure 3.8: Circuit model of a single phase permanent magnet machine.

where

$$v_\lambda(t) = \frac{\mathrm{d}\lambda}{\mathrm{d}t} \tag{3.10}$$

Once again, integrating the instantaneous power yields the total electrical input energy to the system.

$$E_e = \int_0^T p_e(t)\mathrm{d}t \tag{3.11}$$

If no losses are considered and the change in internal energy of the system over a cycle is taken as zero, the law of conservation of energy requires that

$$E_m = -E_e \tag{3.12}$$

where the minus is a result of calculating both the mechanical and electrical power as input power. Thus, the procedure for this accuracy check is as follows:

- Calculate E_m using (3.7) and (3.8)
- Calculate E_e using (3.9), (3.10) and (3.11)
- Calculate the relative error $e = \left|\frac{E_m+E_e}{E_e}\right| \times 100$

If the error is too large, it may be an indication that the mesh is not fine enough in all areas of the model or that some other calculation is incorrect.

3.9 Conclusions

In this chapter several additions and improvements over the original package were discussed. The improvements all contribute to providing the user with a tool that is easier to work with and more powerful. There are three major improvements still to be discussed in the chapters that follow. These are the addition of multiple air-gap capabilities, the simulation of class III problems and parallelisation.

Chapter 4

Relative movement

4.1 Introduction

In time-stepped finite element simulations of electrical machines, it is required that different parts of the finite element model move relative to each other. This chapter is concerned with how this relative movement is accomplished. A brief overview of several movement handling techniques is given in section 4.2. Thereafter, the focus shifts to the movement handling techniques used in *SEMFEM*. The implementation of a simple air-gap mesher is discussed in section 4.3 and the addition of multiple air-gap capabilities is discussed in section 4.4.

4.2 Overview of movement handling techniques

There are several different methods of handling the movement between different parts of the mesh. The most prominent of these methods are described here briefly.

Moving-band
: This technique is based on remeshing of the air-gap region for different positions of the moving component, while the rest of the mesh stays constant. The terms in the stiffness matrix affected by the nodes in the air-gap region are recalculated for different positions. See the paper by Davat et al. [6] for details. A primitive version of this method was implemented in the program during the development of the 2D axisymmetric solver.

Sliding-surface
: This technique is based on joining different parts of the mesh on an interface where

nodes need not coincide, but can slide freely. This method has the advantage that any remeshing is avoided and that only the terms of the stiffness matrix affected by nodes on the surface of movement need to be recalculated for different positions. See the paper by Rodger et al. [21] for details. This method was not implemented in the program.

Air-gap element

This technique was first proposed by Abdel-Razek et al. [2] and is the main method used in the program. Different formulations of the air-gap element is required for the three problem classes. As mentioned in section 2.6, air-gap elements for problem classes I and II have been implemented by previous contributors. The derivation of an air-gap element for problem class III however, has never been carried out before. This derivation, carried out by the author, is documented in section 5.3.

Spectral element

This technique is based on a reformulation of the air-gap element, allowing the use of specialised solvers which reduce the computational time to such an extent that the method becomes competitive with the moving-band and sliding-surface techniques. See the paper by De Gersem and Weiland [11] for details. It is recommended that this technique is investigated in the future as it may be beneficial in terms of computational efficiency, although, at first glance it appears that the implementation of this technique may necessitate significant modifications to some of the subsystems of the program. The gains may not be sufficient to warrant the development cost.

4.3 Air-gap mesher

A simple air-gap mesher, based on the moving-band technique, was implemented in the program. The primary motivation for doing this was to enable thorough testing of the 2D axisymmetric solver prior to the implementation of the axisymmetric air-gap element (see chapter 5). An example of a mesh that has been moved using this technique is shown in figure 4.1. As is clear in the figure, the mesh is poorly connected at the edges of the mesh. Thus, this implementation will not suffice for models with periodic boundary conditions. For linear machines where the entire machine is modelled and a surrounding air box is used, such as the linear machines considered in chapter 7, this method is acceptable.

A flow diagram of the program using the air-gap mesher is shown in figure 4.2. The stator and rotor or translator mesh is first generated in the usual fashion with the air-gap not meshed. This mesh is repeatedly used as a foundation. The rotor or translator is then moved to the desired position at each time-step and the air-gap mesh at this position is

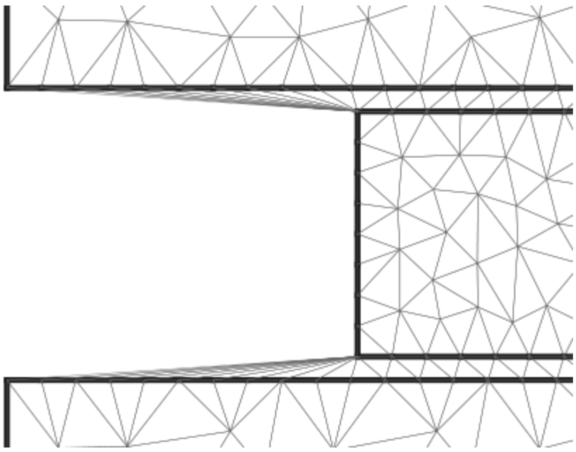


Figure 4.1: Movement handling using the simple air-gap mesher

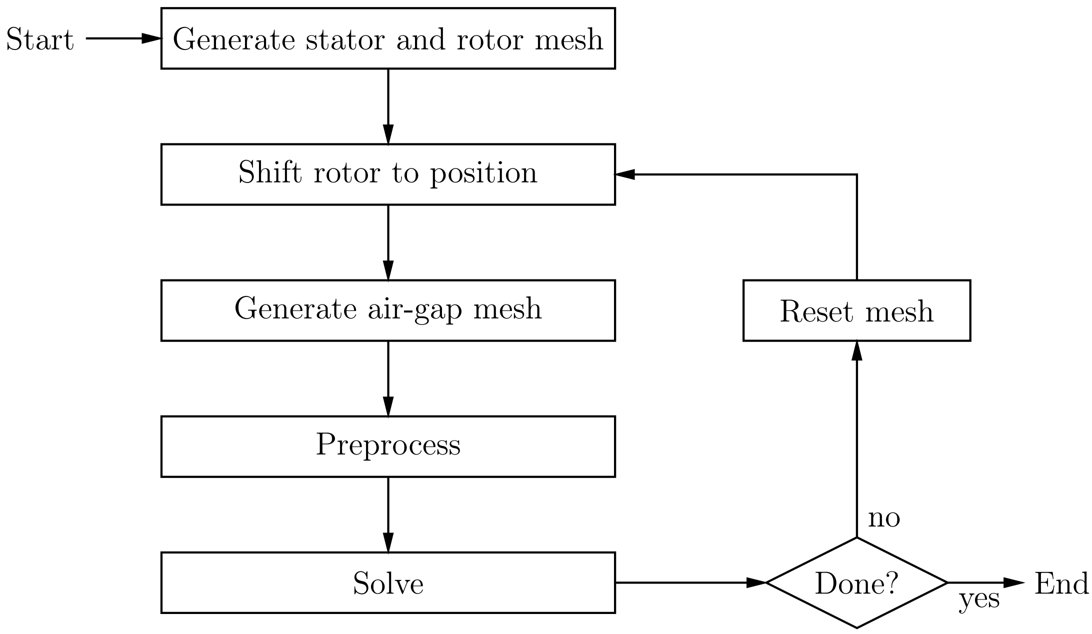


Figure 4.2: Flow diagram of a simulation using the simple air-gap mesher

generated, whereafter the solution at the specific position is calculated. This process is repeated until all positions have been solved.

This method was quite satisfactory for the purpose of testing the axisymmetric solver, which required a quick implementation of an air-gap mesher without a concern for performance. However, it is inefficient in the sense that the preprocessing operation, which is quite expensive, is repeated many times. It is certainly possible to avoid many of the redundant calculations in this method.

Most, if not all, commercial finite element simulation packages that are used to simulate electrical machines mesh the air-gap region. In the future, it may be worthwhile to implement a proper movement handling scheme based on either the moving-band or sliding-surface techniques. Even though the comparison by Gerber et. al. [10] mentioned in the following section showed that the performance of this program was comparable to that of commercial packages, more definitive results on the performance of different movement handling schemes could be obtained with a single package implementing all the techniques mentioned earlier.

4.4 Multiple air-gap elements

A big limitation of the original version of the program was that it was only capable of analysing models with a single air-gap using the air-gap element method. This prohibited the use of the program in the analysis of many machine topologies. This limitation was addressed during the course of this work.

The greatest challenge in adding multiple air-gap functionality to the program lay in understanding how air-gap elements are coupled to the rest of the finite element mesh and identifying the pieces of code that implement this coupling. Mathematically, the coupling of air-gap elements to the traditional mesh can be expressed as

$$K_{ij} = \sum K^e_{mn} + \sum_{k=1}^{N_G} \sum K^{\varepsilon k}_{mn} \tag{4.1}$$

where the first term represents contributions from the traditional mesh and the second term represents the contributions from a total of N_G air-gaps in comparison to (2.12) where only a single air-gap is considered. When using the Newton-Raphson method described in section 2.7.3, the addition of multiple air-gaps leads to the general term of the global Jacobian matrix being given by

$$\begin{aligned} J_{ij} &= \sum J^e_{mn} + \sum_{k=1}^{N_G} \sum J^{\varepsilon k}_{mn} \\ &= \sum J^e_{mn} + \sum_{k=1}^{N_G} \sum K^{\varepsilon k}_{mn} \end{aligned} \tag{4.2}$$

In the above equation, the local Jacobian matrix of an air-gap element, $\mathbf{J}^{\varepsilon k}$, simply evaluates to the local stiffness matrix of the air-gap element. This is because there is only air in the air-gap element and so the permeability – and therefore the coefficients of the air-gap element stiffness matrix – is independent of the solution vector $\mathbf{u}$. Once (4.2) was mastered, modifying the code to allow multiple air-gaps was a relatively simple matter.

The modification of the original program to allow models with multiple air-gaps to be analysed, was done in the following manner: Firstly, additional variables were declared to provide storage for the additional air-gap element data. Secondly, the preprocessor was modified to setup more than one air-gap correctly. Thirdly, the solver was modified to take the contribution to the stiffness matrix coefficients from all air-gap elements into account. Lastly, the torque or force calculations were modified to take the contribution from both neighbouring air-gap elements into account.

Listing 4.1 shows the implementation of the calculation of the global Jacobian matrix, according to (4.2). In the first part of the listing the contributions from traditional elements is added and in the second part, the contributions from the air-gap elements are added.

Figure 4.3 shows a rotating machine with two air-gaps. Note that in order to move both parts of the rotor in the same direction, the air-gap elements have to be shifted in opposite directions. When the torque is calculated, the contributions should be subtracted from each other to yield the total torque. This is valid if the air-gap elements are setup so that they have the same orientation, i.e. upper edges are further away from the origin than lower edges. The total torque (action and reaction) is then given by

$$T_a = T_{1a} - T_{2a} \qquad T_r = T_{1r} - T_{2r} \tag{4.3}$$

Listing 4.2 shows the implementation of the torque calculation for the case of two air-gaps, according to (4.3). The `torque` subroutine calculates the torque contribution from a single air-gap element.

At first, the program was adapted to make provision for a second air-gap element. The accuracy and performance of the program was then evaluated by comparison against commercially available finite element simulation packages by Gerber et. al. [10]. The results, some of which will be presented in chapter 7, showed that the package is a viable alternative to the commercial packages in terms of accuracy and performance.

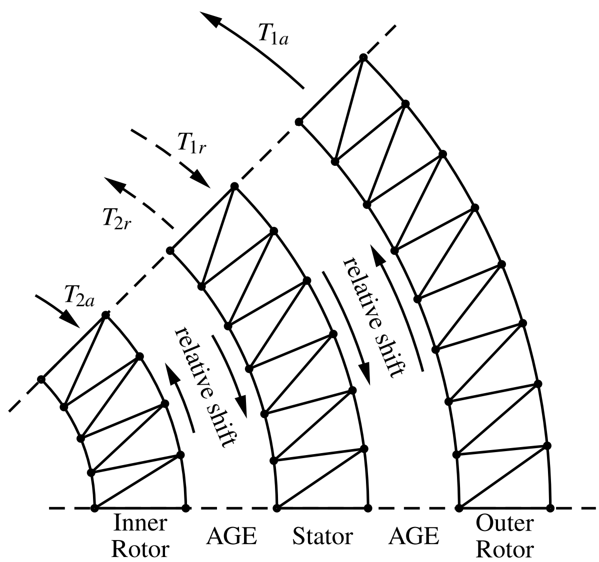


Figure 4.3: Movement and torque calculation for a double air-gap model

```
! Traditional mesh
do 97 ne=1,nelm
    .
    .
    .
    do 94 j=1,3
        ! this is the bit where the jacobian terms gets inserted
        j1=node(ne,j)
        if (nd(node(ne,j)).eq.0) goto 94
        icol=iabs(nd(node(ne,j)))
        jj=isign(1,nd(j1))
        if (icol.ge.irow) then
        k=jdiag(icol)+irow-icol
        !s see binns p. 297 eq. (10.41)
        st(k)=st(k)+ii*jj*(rel(ne)*s(ne,i,j) + &
            2.0d0/ar(ne)*grel(ne)*u(i)*u(j))
        end if
94          continue
95        continue
97    continue

! Air-gap elements
if (air_gap_method >= 1) then
    do age_i = 1, air_gap_method
        do 81 i=1, ages(age_i)%nraz
            uraz(i)=0.0d0
            do inr = 1, ages(age_i)%nraz
                uraz(i)=uraz(i)+ages(age_i)%sraz(i,inr)* &
                    aold(ages(age_i)%razind(inr))
            end do
            if (ages(age_i)%razpnt(i).eq.0) goto 81
            irow = iabs(ages(age_i)%razpnt(i))
            ii=isign(1,ages(age_i)%razpnt(i))
            r(irow)=r(irow)+uraz(i)*v0
            do 82 j=1,ages(age_i)%nraz
                if (ages(age_i)%razpnt(j).eq.0) goto 82
                icol = iabs(ages(age_i)%razpnt(j))
                jj=isign(1, ages(age_i)%razpnt(j))
                if (icol.ge.irow) then
                    k=jdiag(icol)+irow-icol
                    st(k)=st(k)+ii*jj*ages(age_i)%sraz(i,j)*v0
                end if
82          continue
81      continue
    end do
end if
```

Listing 4.1: Code in the non-linear solver implementing the calculation of the coefficients of the global Jacobian matrix according to (4.2) for problem classes I and II.

```
call torque(ages(1)%bot_gap, ages(1)%top_gap, temp_torq, 481, np, &
     ages(1)%nrs, ages(1)%nraz, w, m%a, &
     ages(1)%an, ages(1)%bn, ages(1)%razind, n_ptch)

torq_vec(step_i+1) = temp_torq

call torque(ages(2)%bot_gap, ages(2)%top_gap, temp_torq, 481, np, &
     ages(2)%nrs, ages(2)%nraz, w, m%a, &
     ages(2)%an, ages(2)%bn, ages(2)%razind, n_ptch)

torq_vec(step_i+1) = torq_vec(step_i+1) - temp_torq
```

Listing 4.2: Code implementing the calculation of torque for a two air-gap model, according to (4.3).

Later on, the program was further enhanced by making provision for any number of air-gaps. This enabled topologies such as the combined magnetic gearbox and electrical machine (see section 7.5) to be simulated using this program. To the author's knowledge, *SEMFEM* is the only implementation capable of simulating machines with multiple air-gaps using the air-gap element method.

Chapter 5

2D axisymmetric solver

5.1 Introduction

One of the primary goals of this work was to allow the simulation of tubular axisymmetric machines using the *SEMFEM* package. In this section, the process of adding this functionality to the program will be discussed. Firstly, a derivation of the system equation coefficients for the axisymmetric case will be given in section 5.2. This derivation follows the same thread as the derivation of the Cartesian system equation coefficients given in section 1.2.2. Secondly, the derivation of an axisymmetric air-gap element that can be used to facilitate movement in class III problems will be given in section 5.3. Thirdly, the coefficients of the Jacobian matrix used in the Newton-Raphson method will be derived for the axisymmetric case in section 5.4. In section 5.5, the derivation of a formula used to calculate the force on the translator, in the direction of movement, using the axisymmetric air-gap element is presented. The modification of the calculation of flux linkage is discussed in section 5.6. Finally, the reader is referred to appendix D where the code implementing the calculation of force in the axisymmetric air-gap element is given.

In the derivations to come, integrals will arise that are not readily evaluated analytically. In these cases, the integrals are evaluated using a numerical integration scheme, namely Gaussian quadrature. Binns et. al. [3] recommended that this technique is used. A short discussion on this integration method is presented in appendix E.

5.2 General system equation coefficients

Here follows a derivation of the system equation coefficients for the axisymmetric magnetostatic problem under discussion, described as problem class III in section 1.4.2. A more detailed derivation is given in appendix A. Although expressions for the system equation coefficients are presented by Binns et. al. [3] and could have been used as is, this derivation is given for completeness sake and also to provide added clarity. This derivation is similar to that of Binns et. al. [3], but it differs in some fine points of notation.

The problem of solving (1.1) in the problem domain Ω is approached in the usual fashion using the weighted residual method,

$$\int_\Omega \mathbf{w_i} \cdot \mathbf{R} \mathrm{d}\Omega = 0 \tag{5.1}$$

$$\int_\Omega \mathbf{w_i} \cdot \left(\nabla \times \frac{1}{\mu} (\nabla \times \mathbf{A}) \right) \mathrm{d}\Omega = \int_\Omega \mathbf{w_i} \mathbf{J} \mathrm{d}\Omega \tag{5.2}$$

Even though the solution variable $\mathbf{A}$ will only have one component A_ϕ, the above equations are given in vector form. The reason being that in this cylindrical case, the expression $\nabla \times \frac{1}{\mu}(\nabla \times \mathbf{A})$ does not simplify to $\nabla \cdot \frac{1}{\mu} \nabla A_\phi$ as in section 1.2.2. Thus, Green's first theorem (see Binns et. al. [3, p. 454]) cannot be used to weaken the continuity conditions on A_ϕ. However, a vector form of this theorem can be used. This theorem states that

$$\int_\Omega \mathbf{F} \cdot \nabla \times \mathbf{G} \mathrm{d}\Omega = \int_\Omega \mathbf{G} \cdot \nabla \times \mathbf{F} \mathrm{d}\Omega - \int_\Gamma (\mathbf{F} \times \mathbf{G}) \cdot \mathbf{n} \mathrm{d}\Gamma \tag{5.3}$$

By identifying $\mathbf{F} = \mathbf{w_i}$ and $\mathbf{G} = \frac{1}{\mu}(\nabla \times \mathbf{A})$, (5.2) can be transformed into

$$\int_\Omega (\nabla \times \mathbf{w_i}) \cdot \frac{1}{\mu} (\nabla \times \mathbf{A}) \mathrm{d}\Omega - \int_\Gamma \left(\mathbf{w_i} \times \frac{1}{\mu} (\nabla \times \mathbf{A}) \right) \cdot \mathbf{n} \mathrm{d}\Gamma = \int_\Omega \mathbf{w_i} \mathbf{J} \mathrm{d}\Omega \tag{5.4}$$

The second term on the left vanishes for all cases considered in this work (see appendix A for details). Expanding the integrands in the above equation and dropping the integration over $\mathrm{d}\phi$ yields

$$\int_\Omega \frac{1}{\mu} \left(\frac{\partial \omega_i}{\partial z} \frac{\partial A_\phi}{\partial z} + \frac{1}{r^2} \frac{\partial (r\omega_i)}{\partial r} \frac{\partial (rA_\phi)}{\partial r} \right) r \mathrm{d}r \mathrm{d}z = \int_\Omega \omega_i J_\phi r \mathrm{d}r \mathrm{d}z \tag{5.5}$$

At this stage the finite element approximation,

$$A_\phi = \sum_{i=1}^{3} N_i u_i^\varepsilon \tag{5.6}$$

is introduced, with the shape function N_i defined as

$$N_i = \frac{a_i + b_i r + c_i z}{2A} \qquad \begin{aligned} a_1 &= r_2 z_3 - r_3 z_2 \\ b_1 &= z_2 - z_3 \\ c_1 &= r_3 - r_2 \end{aligned} \tag{5.7}$$

Substituting (5.6) into (5.5), considering only a single element yields

$$\int_{\Omega^e} \frac{1}{\mu} \sum_{j=1}^{3} \left[\left(\frac{\partial \omega_i}{\partial z} \frac{\partial N_j}{\partial z} + \frac{1}{r^2} \frac{\partial (r\omega_i)}{\partial r} \frac{\partial (rN_j)}{\partial r} \right) r u_j^\varepsilon \right] \mathrm{d}r \mathrm{d}z = \int_{\Omega^e} \omega_i J_\phi r \mathrm{d}r \mathrm{d}z \tag{5.8}$$

If Galerkin weighting is applied ($\omega_i = N_i$), three of the above equations can be assembled into a matrix equation of the form

$$\mathbf{K}^e \mathbf{u}^e = \mathbf{f}^e \tag{5.9}$$

with the coefficients of the stiffness matrix $\mathbf{K}^e$ and the forcing vector $\mathbf{f}^e$ given by

$$K_{ij}^e = \int_{\Omega^e} \frac{1}{\mu} \left(\frac{\partial N_i}{\partial z} \frac{\partial N_j}{\partial z} + \frac{1}{r^2} \frac{\partial (rN_i)}{\partial r} \frac{\partial (rN_j)}{\partial r} \right) r \mathrm{d}r \mathrm{d}z \tag{5.10}$$

and

$$f_i^e = \int_{\Omega^e} N_i J_\phi r \mathrm{d}r \mathrm{d}z \tag{5.11}$$

Substituting (5.7) into (5.10) and (5.11) and evaluating the partial derivatives result in the system equation coefficients being given by

$$K_{ij}^e = \int_{\Omega^e} \frac{1}{4A^2 \mu} \left[c_i c_j + \frac{1}{r^2} (a_i + 2b_i r + c_i z)(a_j + 2b_j r + c_j z) \right] r \mathrm{d}r \mathrm{d}z \tag{5.12}$$

and

$$f_i^e = \int_{\Omega^e} J_\phi \left(\frac{a_i + b_i r + c_i z}{2A} \right) r \mathrm{d}r \mathrm{d}z \tag{5.13}$$

which is typically evaluated with a numerical integration scheme such as Gaussian quadrature.

5.3 Axisymmetric air-gap element

Having gone through the derivation of the axisymmetric system equation coefficients and implementing these calculations successfully, the program was now capable of simulating class III problems without movement. Problems with motion could also be simulated using the simple air-gap mesher described in section 4.3, but this was not satisfactory because it did not allow analytical force calculation and the simple implementation of the air-gap mesher was inefficient. An air-gap element for class III problems was required. To the knowledge of the author, the axisymmetric air-gap element's stiffness matrix coefficients have never been derived before. These coefficients were derived by the author, observing the procedure followed by Wang [30] in the derivation of the Cartesian air-gap element's stiffness matrix coefficients. The complete derivation is given in appendix C. Here follows a summary thereof.

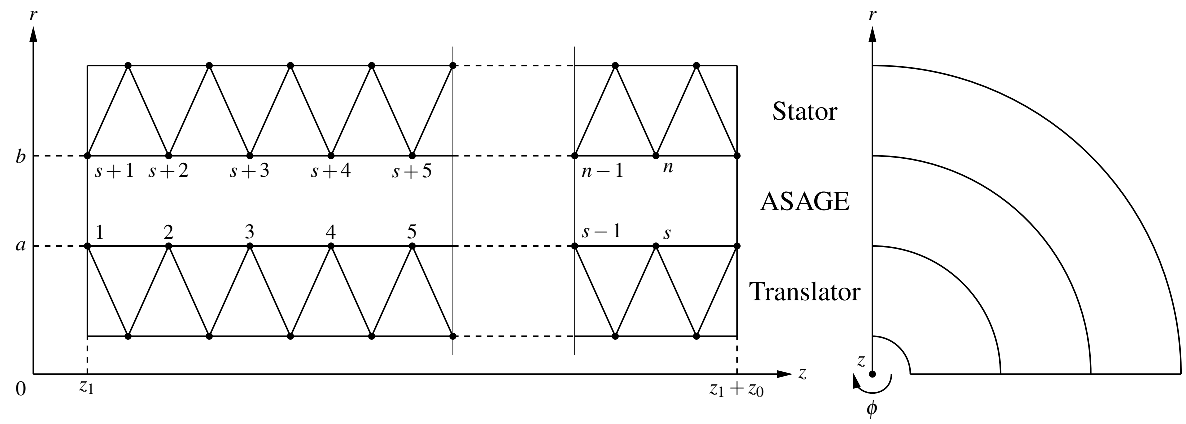


Figure 5.1: The domain of the axisymmetric air-gap element

5.3.1 Air-gap field solution

In an axisymmetric air-gap region, such as that illustrated in figure 5.1, the magnetic vector potential A_ϕ is governed by

$$\nabla \times (\nabla \times \mathbf{A}) = 0 \tag{5.14}$$

$$\frac{\partial^2 A_\phi}{\partial z^2} + \frac{\partial^2 A_\phi}{\partial r^2} + \frac{1}{r}\frac{\partial A_\phi}{\partial r} - \frac{1}{r^2}A_\phi = 0 \tag{5.15}$$

For the purpose of the air-gap element the following boundary conditions will be imposed on the solution of (5.15),

$$A_\phi(r, z) = A_\phi(r, z + z_0) \tag{5.16}$$

$$A_\phi(a, z) = \sum_i \alpha_i(a, z)u_i^\varepsilon \tag{5.17}$$

$$A_\phi(b, z) = \sum_i \alpha_i(b, z)u_i^\varepsilon \tag{5.18}$$

Equation (5.16) represents a positive periodic boundary condition at the sides of the air-gap element while (5.17) and (5.18) state that the solution on the edge of the air-gap element should conform to the solution on the edge of the traditionally meshed regions of the model. Using the technique of separation of variables a solution to (5.15) that satisfies (5.16) can be obtained. This solution is

$$A_\phi(r, z) = B_0 r + C_0 r^{-1} + \sum_{n=1}^{\infty}(D_n \cos\lambda_n z + E_n \sin\lambda_n z)(F_n I_1(\lambda_n r) + G_n K_1(\lambda_n r)) \tag{5.19}$$

with

$$\lambda_n = \frac{2\pi n}{z_0} \quad , \quad n \in \mathbf{Z} \tag{5.20}$$

In order to simplify the satisfaction of (5.17) and (5.18), $A_\phi(r, z)$ is expressed as

$$A_\phi(r, z) = A_{\phi_1}(r, z) + A_{\phi_2}(r, z) \tag{5.21}$$

and the boundary conditions on $A_{\phi_1}(r,z)$ and $A_{\phi_2}(r,z)$ are defined as

$$A_{\phi_1}(a,z) = A_\phi(a,z) \tag{5.22}$$
$$A_{\phi_1}(b,z) = 0 \tag{5.23}$$

and

$$A_{\phi_2}(a,z) = 0 \tag{5.24}$$
$$A_{\phi_2}(b,z) = A_\phi(b,z) \tag{5.25}$$

In this way, the sum of $A_{\phi_1}(r,z)$ and $A_{\phi_2}(r,z)$ have boundary values equal to that of $A_\phi(r,z)$, but the boundary conditions imposed on $A_{\phi_1}(r,z)$ and $A_{\phi_2}(r,z)$ are satisfied more easily.

The function $A_{\phi_1}(r,z)$ is defined similar to (5.19)

$$A_{\phi_1}(r,z) = B_{10}r + C_{10}r^{-1} + \sum_{n=1}^{\infty}(D_{1n}\cos\lambda_n z + E_{1n}\sin\lambda_n z)(F_{1n}I_1(\lambda_n r) + G_{1n}K_1(\lambda_n r)) \tag{5.26}$$

Setting $A_{\phi_1}(b,z)$ equal to zero in order to satisfy (5.23) and by defining a new set of coefficients, it can be shown that the expression for $A_{\phi_1}(r,z)$ can be simplified to

$$A_{\phi_1}(r,z) = a_{10}\left(r - \frac{b^2}{r}\right) + \sum_{n=1}^{\infty}\left(I_1(\lambda_n r) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)}K_1(\lambda_n r)\right)(a_{1n}\cos\lambda_n z + b_{1n}\sin\lambda_n z) \tag{5.27}$$

Following the same procedure to satisfy (5.24), $A_{\phi_2}(r,z)$ is given by

$$A_{\phi_2}(r,z) = a_{20}\left(r - \frac{a^2}{r}\right) + \sum_{n=1}^{\infty}\left(I_1(\lambda_n r) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)}K_1(\lambda_n r)\right)(a_{2n}\cos\lambda_n z + b_{2n}\sin\lambda_n z) \tag{5.28}$$

Since $A_\phi(a,z) = A_{\phi_1}(a,z)$ and $A_\phi(b,z) = A_{\phi_2}(b,z)$, the solution on the boundaries at $r=a$ and $r=b$ is

$$A_\phi(a,z) = a_{10}\left(a - \frac{b^2}{a}\right) + \sum_{n=1}^{\infty}\left(I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)}K_1(\lambda_n a)\right)(a_{1n}\cos\lambda_n z + b_{1n}\sin\lambda_n z) \tag{5.29}$$

$$A_\phi(b,z) = a_{20}\left(b - \frac{a^2}{b}\right) + \sum_{n=1}^{\infty}\left(I_1(\lambda_n b) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)}K_1(\lambda_n b)\right)(a_{2n}\cos\lambda_n z + b_{2n}\sin\lambda_n z) \tag{5.30}$$

The next step is to express the solution on the boundaries at $r=a$ and $r=b$ in terms of the vector potential at the nodes on these boundaries. This expression should take a similar form to (5.29) and (5.30) in order to allow coefficients to be equated. Wang [30] showed in his derivation of the CAGE that the solution on the boundaries of the CAGE at $y=a$ and $y=b$ could be expressed in terms of a Fourier series and the nodal values on the boundary. Although this derivation was done in the Cartesian coordinate system, it is also valid in the cylindrical coordinate system used with the ASAGE with x equivalent to z and y equivalent to r. The resulting forms of equation (5.17) and (5.18) are

$$A_\phi(a,z) = \sum_{i=1}^{s} u_i^\varepsilon \left[\frac{1}{2}a_{0i} + \sum_{n=1}^{\infty}(a_{ni}\cos\lambda_n z + b_{ni}\sin\lambda_n z)\right] \tag{5.31}$$

and

$$A_\phi(b,z) = \sum_{j=s+1}^{t} u_j^\varepsilon \left[\frac{1}{2}a_{0j} + \sum_{n=1}^{\infty}(a_{nj}\cos\lambda_n z + b_{nj}\sin\lambda_n z)\right] \tag{5.32}$$

where the coefficients a_{0i}, a_{ni} and b_{ni} are functions of λ_n, z_0 and the z-coordinates of the nodes. Through comparison of the coefficients of (5.29) and (5.31), it can be seen that

$$\begin{aligned} a_{10}\left(a - \frac{b^2}{a}\right) &= \sum_{i=1}^{s} u_i^\varepsilon \frac{1}{2} a_{0i} \\ a_{10} &= \frac{1}{2(a-\frac{b^2}{a})}\sum_{i=1}^{s} a_{0i}u_i^\varepsilon \end{aligned} \tag{5.33}$$

and

$$\begin{aligned} a_{1n}\left(I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1\lambda_n b}K_1(\lambda_n a)\right) &= \sum_{i=1}^{s} a_{ni}u_i^\varepsilon \\ a_{1n} &= \frac{\sum_{i=1}^{s} a_{ni}u_i^\varepsilon}{I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)}K_1(\lambda_n a)} \end{aligned} \tag{5.34}$$

By similar comparisons

$$a_{2n} = \frac{\sum_{j=s+1}^{t} a_{nj}u_j^\varepsilon}{I_1(\lambda_n b) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)}K_1(\lambda_n b)} \tag{5.35}$$

$$b_{1n} = \frac{\sum_{i=1}^{s} b_{ni}u_i^\varepsilon}{I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)}K_1(\lambda_n a)} \tag{5.36}$$

$$b_{2n} = \frac{\sum_{j=s+1}^{t} b_{nj}u_j^\varepsilon}{I_1(\lambda_n b) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)}K_1(\lambda_n b)} \tag{5.37}$$

Substituting (5.33) through (5.37) back into (5.27) and (5.28), (5.21) becomes

$$A_\phi(r,z) = \sum_{i=1}^{t} \alpha_i^\varepsilon(r,z)u_i^\varepsilon \tag{5.38}$$

with

$$\alpha_i^\varepsilon(r,z) = \frac{r - \frac{c_i^2}{r}}{c_i' - \frac{c_i^2}{c_i'}}\frac{a_{0i}}{2} + \sum_{n=1}^{\infty}\left(\frac{I_1(\lambda_n r) - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)}K_1(\lambda_n r)}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)}K_1(\lambda_n c_i')}\right)(a_{ni}\cos\lambda_n z + b_{ni}\sin\lambda_n z) \tag{5.39}$$

$$\text{where}\begin{cases} c_i = b \quad \text{and} \quad c_i' = a \quad \text{if} \quad i\epsilon\{1,2,\ldots,s\} \\ c_i = a \quad \text{and} \quad c_i' = b \quad \text{if} \quad i\epsilon\{s+1,\ldots,t\} \end{cases} \tag{5.40}$$

5.3.2 Stiffness matrix

In the axisymmetric case under consideration, the weighted residual method gives the following formula for the minimisation of the residual,

$$\int_{\Omega^\epsilon} \left(\frac{\partial \omega_i}{\partial z} \frac{\partial A_\phi}{\partial z} + \frac{1}{r^2} \frac{\partial (r\omega_i)}{\partial r} \frac{\partial (rA_\phi)}{\partial r} \right) \mathrm{d}\Omega^\epsilon = 0 \tag{5.41}$$

with ω_i an arbitrary weighting function and Ω^ϵ the domain of the air-gap element. Using Galerkin weighting, the weighting functions are chosen as the shape functions ($\omega_i = \alpha_i^\varepsilon$) and by substituting (5.39) into (5.41) the expression for the i'th row of the system equation becomes

$$R_i = \int_{\Omega^\epsilon} \sum_{j=1}^{t} \left[\left(\frac{\partial \alpha_i^\varepsilon}{\partial z} \frac{\partial \alpha_j^\varepsilon}{\partial z} + \frac{1}{r^2} \frac{\partial (r\alpha_i^\varepsilon)}{\partial r} \frac{\partial (r\alpha_j^\varepsilon)}{\partial r} \right) u_j^\varepsilon \right] \mathrm{d}\Omega^\epsilon = 0 \tag{5.42}$$

resulting in the general term of the air-gap element's stiffness matrix being given by

$$K_{ij}^\epsilon = \int_{\Omega^\epsilon} \left(\frac{\partial \alpha_i^\varepsilon}{\partial z} \frac{\partial \alpha_j^\varepsilon}{\partial z} + \frac{1}{r^2} \frac{\partial (r\alpha_i^\varepsilon)}{\partial r} \frac{\partial r\alpha_j^\varepsilon}{\partial r} \right) r\mathrm{d}r\mathrm{d}z \tag{5.43}$$

Evaluating the derivatives and simplifying, the following expression for K_{ij}^ϵ is obtained

$$\begin{aligned} K_{ij}^\epsilon = & \frac{z_0(b^2 - a^2)}{2(c_i' - \frac{c_i^2}{c_i'})(c_j' - \frac{c_j^2}{c_j'})} \cdot a_{0i} a_{0j} \\ & + \frac{z_0}{2} \sum_{n=1}^{\infty} (a_{ni}a_{nj} + b_{ni}b_{nj}) \int_a^b \left[\lambda_n^2 f_{2i}(r) f_{2j}(r) r\mathrm{d}r + \frac{1}{r^2} f_{4i}(r) f_{4j}(r) \right] r\mathrm{d}r \end{aligned} \tag{5.44}$$

with the functions $f_{2i}(r)$ and $f_{4i}(r)$ defined as

$$f_{2i}(r) = \left(\frac{I_1(\lambda_n r) - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n r)}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n c_i')} \right) \tag{5.45}$$

$$f_{4i}(r) = r\lambda_n \left(\frac{I_0(\lambda_n r) + \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_0(\lambda_n r)}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n c_i')} \right) \tag{5.46}$$

The integral in (5.44) is evaluated numerically and the infinite series is truncated at some point.

During testing of the axisymmetric air-gap element, problems were experienced with (5.45) and (5.46). For large values of the argument, the modified Bessel functions I_n and K_n become very large and very small respectively. The size of the argument is a function of $\lambda_n = \frac{2\pi n}{z_0}$. Referring to (5.44), the solution is obviously more accurate when more terms are used in the sum that should theoretically go to infinity, but for large values of n, the arguments of the modified Bessel functions become large and the ratio $\frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)}$ becomes a large number divided by a small number resulting in a very large number, so much so that overflow errors occur. These errors put a limit on the amount of terms that could be used and so also on the accuracy of the resulting air-gap field solution. The problem can be alleviated by expressing (5.45) and (5.46) in an alternative manner, namely

$$f_{2i}(r) = \left(\frac{I_1(\lambda_n r)K_1(\lambda_n c_i) - I_1(\lambda_n c_i)K_1(\lambda_n r)}{I_1(\lambda_n c_i')K_1(\lambda_n c_i) - I_1(\lambda_n c_i)K_1(\lambda_n c_i')} \right) \tag{5.47}$$

and

$$f_{4i}(r) = r\lambda_n \left(\frac{I_0(\lambda_n r)K_1(\lambda_n c_i) + I_1(\lambda_n c_i)K_0(\lambda_n r)}{I_1(\lambda_n c_i')K_1(\lambda_n c_i) - I_1(\lambda_n c_i)K_1(\lambda_n c_i')} \right) \tag{5.48}$$

in which the products of the modified Bessel functions are large numbers multiplied by small numbers. Although this allowed the use of many more terms before overflow occurred, the individual functions I_n and K_n still caused overflow for larger n. A better solution would be to approximate these products directly, thus avoiding excessively large or small terms. This solution was however not implemented but it is a recommended improvement for future development.

5.4 Axisymmetric Newton-Raphson Jacobian

Another modification to the original program that was necessary in order to allow the simulation of class III problems, was the calculation of the coefficients of the Jacobian matrix used in the Newton-Raphson method (see section 2.7.3). A derivation of these coefficients is given in appendix B. The result is that for the axisymmetric case, the coefficients of the local Jacobian matrix are given by

$$J_{ij}^e = K_{ij}^e + \frac{1}{8A^4} \cdot \frac{\partial \kappa}{\partial (p^2)} \cdot \sum_{h=1}^{3} \sum_{k=1}^{3} \left(\int_{\Omega^e} t_{kj} t_{ih} r \mathrm{d}r \mathrm{d}z \right) u_k^e u_h^e \tag{5.49}$$

with

$$t_{ij} = \left[c_i c_j + \frac{1}{r^2}(a_i + 2b_i r + c_i z)(a_j + 2b_j r + c_j z) \right] \tag{5.50}$$

This integral is evaluated numerically using Gaussian quadrature.

5.5 Force calculation

In this section, the calculation of the force on the translator in axisymmetric problems is discussed. This calculation, like the calculation for class I and II problems, is based on the integration of the Maxwell stress tensor on a closed surface surrounding the translator. The surface is chosen to lie in the middle of the air-gap element where the integral can be computed analytically.

According to Stratton [26], the force exerted on a body by a magnetostatic field is given by

$$\mathbf{F} = \oint_{\Gamma} \left[\frac{1}{\mu_0} (\mathbf{B} \cdot \mathbf{n}) \mathbf{B} - \frac{1}{2\mu_0} B^2 \mathbf{n} \right] \mathrm{d}a \tag{5.51}$$

with Γ a closed surface surrounding the body and $\mathbf{n}$ the unit outward normal vector to this surface. As explained in section 2.8, only the part of the surface integral in the air-gap contributes to the force. On this part of the surface, $\mathbf{n} = \mathbf{i}_r$. Considering only this part of the closed surface and expanding the vectors into their components, (5.51) becomes

$$\begin{bmatrix} F_r \\ F_\phi \\ F_z \end{bmatrix} = \int_{\Gamma^\varepsilon} \frac{1}{\mu_0} \begin{bmatrix} B_r \\ B_\phi \\ B_z \end{bmatrix} \left(\begin{bmatrix} B_r \\ B_\phi \\ B_z \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) - \frac{1}{2\mu_0} B^2 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \mathrm{d}a \tag{5.52}$$

Usually, one is most interested in the the force in the direction of movement F_z, given by

$$\begin{aligned} F_z &= \int_{\Gamma^\varepsilon} \frac{1}{\mu_0} B_r B_z \mathrm{d}a \\ &= \frac{1}{\mu_0} \int_{\phi=0}^{2\pi} \int_{z=z_1}^{z_1+z_0} B_r B_z r \mathrm{d}z \mathrm{d}\phi \\ &= \frac{2\pi}{\mu_0} \int_{z_1}^{z_1+z_0} B_r B_z r \mathrm{d}z \end{aligned} \tag{5.53}$$

The components B_r and B_z is obtained by evaluating $\mathbf{B} = \nabla \times \mathbf{A}$ in cylindrical coordinates

$$B_r = -\frac{\partial A_\phi}{\partial z} \qquad B_z = \frac{1}{r} \frac{\partial (r A_\phi)}{\partial r} \tag{5.54}$$

Substituting the solution of A_ϕ in the air-gap in (5.38) into the above, the components become

$$B_r = -\sum_{i=1}^{t} \frac{\partial \alpha_i}{\partial z} u_i^\varepsilon \qquad B_z = \frac{1}{r} \sum_{i=1}^{t} \frac{\partial (r\alpha_i)}{\partial r} u_i^\varepsilon \tag{5.55}$$

with u_i the vector potential at the i'th node on the boundary of the air-gap element. Using the above, (5.53) becomes

$$F_z = -\frac{2\pi}{r\mu_0} \sum_{i=1}^{t} \sum_{j=1}^{t} u_i^\varepsilon u_j^\varepsilon \int_{z_1}^{z_1+z_0} \frac{\partial \alpha_i}{\partial z} \frac{\partial (r\alpha_j)}{\partial r} \mathrm{d}z \tag{5.56}$$

From (C.52)

$$\frac{\partial \alpha_i}{\partial z} = \sum_{n=1}^{\infty} f_{2i}(r) g_i'(z) \tag{5.57}$$

and from (C.66)

$$\frac{\partial (r\alpha_i)}{\partial r} = f_{3i}(r) + \sum_{n=1}^{\infty} f_{4i}(r) g_i(z) \tag{5.58}$$

Using these expressions, the integral in (5.56) becomes

$$\begin{aligned} \int_{z_1}^{z_1+z_0} \frac{\partial \alpha_i}{\partial z} \frac{\partial (r\alpha_j)}{\partial r} \mathrm{d}z = \sum_{n=1}^{\infty} \int_{z_1}^{z_1+z_0} & f_{3i}(r) f_{2i}(r) g_i'(z) \mathrm{d}z \\ & + \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} \int_{z_1}^{z_1+z_0} f_{2i}(r) f_{4j}(r) g_i'(z) g_j(z) \mathrm{d}z \end{aligned} \tag{5.59}$$

The first term in (5.59) is zero because of (C.53) and (C.54). Expanding the z dependent part of the second term, (5.59) becomes

$$\int_{z_1}^{z_1+z_0} \frac{\partial \alpha_i}{\partial z} \frac{\partial (r\alpha_j)}{\partial r} \mathrm{d}z = \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} f_{2i}(r) f_{4j}(r) \cdot \int_{z_1}^{z_1+z_0} \lambda_n (b_{ni} \cos \lambda_n z - a_{ni} \sin \lambda_n z)(a_{mj} \cos \lambda_n z + b_{mj} \sin \lambda_n z) \mathrm{d}z \quad (5.60)$$

which, using the identities (C.55) to (C.57), can be simplified to

$$\int_{z_1}^{z_1+z_0} \frac{\partial \alpha_i}{\partial z} \frac{\partial (r\alpha_j)}{\partial r} \mathrm{d}z = \frac{z_0}{2} \sum_{n=1}^{\infty} \lambda_n f_{2i}(r) f_{4j}(r) (b_{ni} a_{nj} - a_{ni} b_{nj}) \quad (5.61)$$

When this expression is substituted back into (5.56), the desired expression for the force in terms of the vector potentials at the boundaries of the air-gap element is obtained, namely

$$F_z = -\frac{z_0 \pi}{r \mu_0} \sum_{i=1}^{t} \sum_{j=1}^{t} u_i^{\varepsilon} u_j^{\varepsilon} \sum_{n=1}^{\infty} \lambda_n f_{2i}(r) f_{4j}(r) (b_{ni} a_{nj} - a_{ni} b_{nj}) \quad (5.62)$$

The implementation of this calculation is presented in appendix D.

5.6 Flux linkage calculation

The flux linkage calculation described in section 2.9 was slightly modified for axisymmetric problems. Specifically, (2.30) becomes

$$\lambda = \frac{2\pi N}{A_C} \left[\sum \frac{\gamma \bar{r} A \sum_{i=1}^{3} u_i}{3} \right] \quad (5.63)$$

where the stack length w has been replaced by $2\pi\bar{r}$ with $\bar{r}$ the average radius of the three vertices of a triangular element. This is only an approximation of the true flux linkage which is given by (2.28), but is was found that this method provides reasonable accuracy. However, it is still recommended that this approximation be replaced by (2.28) and evaluating the integral using Gaussian quadrature.

5.7 Conclusions

All the calculations from chapters 1 and 2 that had to be modified to allow the simulation of axisymmetric problems were documented in this chapter. With these calculations implemented, *SEMFEM* is now capable of simulating many of the tubular topologies used in linear machines. The reader is referred to section 7.4 for a validation of the calculations presented in this chapter.

Chapter 6

Optimisation and parallelisation

6.1 Overview

The task of finding an optimal design for an electrical machine is not a simple one for the following reasons:

- A large number of variables is often needed to describe a machine and the performance of the machine has a complex dependency on these variables.
- In order to accurately model a machine, taking into account non-linearities and complex geometries, finite element analysis must be used. This implies that the evaluation of a single set of design variables is a computationally expensive process. Thus, it is desirable to find an optimal design using as few as possible finite element simulations.

In these circumstances, the most powerful method of optimising a design is numerical optimisation. The advantages of this method are that it can deal with problems involving many design variables, it can handle complex constraints and it only requires information on the objective function and constraints at specific points in the design space. On the other hand, numerical optimisation can never guarantee that the design that it arrives at is *the* optimal design. It is therefore important that users of numerical optimisation do not blindly accept the findings of an optimisation, but critically asses the results and make modifications to the optimisation process where necessary.

Another important issue when using numerical optimisation is the computational cost. When finite element analysis is used, a single function evaluation may be quite expensive and because the optimisation process requires many function evaluations, an optimisation can be a very expensive process.

In the rest of this chapter, the use of *SEMFEM* within optimisation environments and the steps taken to alleviate the problem of high computational cost will be discussed. A few comments on available optimisation strategies will be made in section 6.2. In section 6.3, attention is given to the process of coupling a *SEMFEM* analysis to a specific optimisation program. Lastly, the use of parallel processing to speed the optimisation process is discussed in section 6.4.

6.2 Comments on different optimisation strategies

Optimisation algorithms can generally be divided into two classes: gradient-based algorithms and global algorithms. Generally, gradient-based algorithms reach an optimum using fewer function evaluations than global algorithms. The disadvantage of gradient-based algorithms is that they do not work as well with non-smooth functions and may get stuck at local optima. Global algorithms are not as likely to suffer from these problems, but in general require more function evaluations to converge to an optimum.

Because optimisation algorithm implementations can be rather complex and may require extensive testing and development to reach maturity, it was decided to make use of available optimisation algorithms. Optimisation algorithms from the following vendors were considered:

Scipy: These optimisation algorithms are available as part of the *Scipy* extension of Python.

OpenOpt: These optimisation algorithms are freely available for download from the project's website, http://openopt.org.

Vanderplaats Research & Development develops *VisualDOC* – a powerful optimisation suite which provides many optimisation algorithms.

Although *Scipy* and *OpenOpt* provide several optimisation algorithms, the choices are narrowed down when considering problems with non-linear objective functions and constraints. Without going into an in depth discussion on the merits of the different optimisation algorithms, those provided in *VisualDOC* were selected as the best option. The most important features of these algorithms, which were not always provided by the other algorithms considered, were

- The ability to handle non-linear objective function and constraints.

- Support for hard constraints on the search space. Violation of these hard constraints may typically cause a simulation to crash. Thus, it is important that the optimisation algorithm will never violate them.

- The ability to evaluate the objective function and constraints from a single analysis. Some other algorithms were designed to evaluate these separately, requiring intervention if simulations are to be used efficiently.

Furthermore, *VisualDOC* was designed with support for parallelisation. The implementation of the parallelisation scheme discussed in section 6.4 was possible because *VisualDOC* provides a C application programming interface which was used to develop an MPI application that allows processes to be run in parallel.

VisualDOC provides a wide variety of optimisation algorithms of which only three were used during this work. These algorithms are the modified method of feasible directions, sequential linear programming and sequential quadratic programming. The book by Vanderplaats [28] may be consulted for an explanation of these methods.

6.3 Coupling analysis to optimisation program

6.3.1 Coupling to VisualDOC

In order to use the finite element analysis provided by *SEMFEM* to optimise a machine, the analysis must be coupled to the optimisation program in some way. In figure 1.3, this is illustrated by the optimiser providing the input parameters and accepting the output parameters. In this section, the coupling of *SEMFEM* to *VisualDOC* will be discussed.

If the graphical user interface of *VisualDOC* is used, the exchange of information can be accomplished by variable exchange files. *VisualDOC* typically writes the values of the design variables in a file, `dvar.vef`, before calling the analysis program. The analysis program, *SEMFEM* in this case, then reads this file to obtain the design point to be simulated and writes the output to another file, `resp.vef`, before terminating. *VisualDOC* can then obtain the results from the analysis by reading the `resp.vef` file.

As mentioned in the previous section, *VisualDOC* can also be used via its C application programming interface. When *VisualDOC* is used in this way, the design variables and responses can be passed directly to the *SEMFEM* analysis as parameters of the main analysis function.

6.3.2 General coupling

The fact that *SEMFEM* is based on a scripting approach makes it easy to integrate with almost any optimisation program because a user is free to obtain design variables from, and write output to any exchange files, whatever the desired format may be. Alternatively an analysis function with any function signature can be created as a wrapper function for a *SEMFEM* simulation. The wrapper function can then be passed directly to the optimisation program as a callback function if the optimisation program provides a C or Fortran application programming interface. If the optimisation program is Python based and a callback function is required, the *f2py* tool [20] comes highly recommended. Using this tool, the Fortran callback function can easily be wrapped so as to be callable from Python.

6.4 Parallelisation

6.4.1 Introduction

In section 1.3 the basic principle behind numerical optimisation was explained with reference to figure 1.3. The process illustrated in figure 1.3 is a serial process. If a computer with multiple CPU cores is available, the process can be accelerated by performing some calculations in parallel. There are several areas where possible parallelisation can be exploited to accelerate the optimisation process. The parallelisation of the optimisation process implemented in this work, is illustrated in figure 6.1. Comparing figures 1.3 and 6.1, it can be seen that the serial process has been parallelised in two areas, marked by the grey circles. What happens at each of the circles, can be described as follows:

1. The optimiser requests three simulations simultaneously
2. The time steps of a single simulation is divided and solved on three separate CPU cores, illustrated by the shaded blocks.

Thus, in this example, a total of nine CPU cores can be used simultaneously.

6.4.2 Parallel simulation capabilities

The first area in which parallel simulation capabilities were implemented was in the *SEMFEM* core. This type of parallelisation takes place at the second grey circle in figure 6.1 where the dashed *Finite element simulation* box was enlarged to show the inner

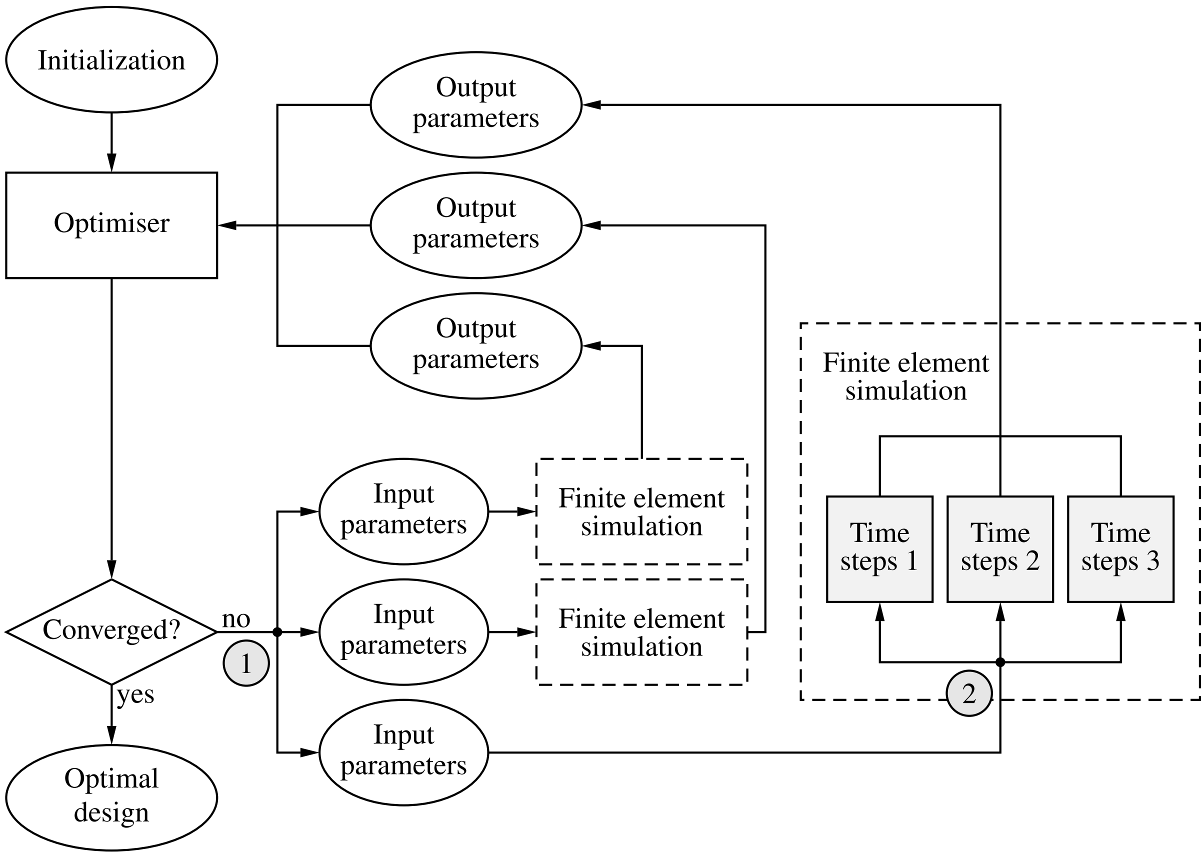


Figure 6.1: Parallel design optimisation using parallel finite element analysis

processes. It works as follows: In a time-stepped finite element simulation, field solutions must be obtained for many different positions of the moving component corresponding to different values of simulation time. This presents an opportunity to use parallel processing because, although the generation of the mesh and some preprocessing calculations cannot easily be performed in parallel, the total time-steps to be solved can easily be distributed among the available CPU cores.

An important requirement of the parallel version of the core library is that the interface exposed to the user must hide the parallelisation as far as possible. In other words, nothing different should be required from the user when running the parallel version as opposed to a serial version. This was largely achieved, but a few minor differences remain. For example, consider listing 6.1 which is an example of the main program of a user simulation. The array `xpar` contains the set of design variables to be analysed. After some initialisation, the `draw` subroutine generates the mesh and the `init_moveloop` subroutine sets the number of time-steps and the currents and positions of the moving component at each time-step. The model is then solved in the `core_solver` subroutine, whereafter some post-processing calculations are performed and the program is terminated cleanly. The only modifications that were necessary to prepare this program for parallel execution was that only the master process (rank 0) should execute the post-processing

```
program single
    use SEMFEM_module
    use mac2_module
    implicit none

    type(mesh) m
    double precision xpar(20), ypar(20)
    double precision out(4)
    integer ierror
    ! Timing
    character(len=10) :: tdate, ttime, tzone
    integer tvalues(10), start_time, end_time

    call date_and_time(tdate, ttime, tzone, tvalues)
    start_time = tvalues(5)*60*60 + tvalues(6)*60 + tvalues(7)

    xpar(1) = 200d-3
    xpar(2) = 20d-3
    xpar(3) = 0.7d0
    xpar(4) = 3d-3
    xpar(5) = 20d-3
    xpar(6) = 0.7d0
    xpar(7) = 10d-3
    xpar(8) = 10d-3
    xpar(9) = 30d-3
    xpar(10) = 35d-3

    call init_config(0)
    call init_input(xpar)
    call draw(m)
    call init_moveloop(m)
    call core_solver(m, ypar)
    if (rank == 0) then
        call postprocess(m)
        out(1) = total_mass
        out(2) = 3000d0 + tpower
        out(3) = 95d0 - eff
        out(4) = translator_mass - 3d0
    end if
    call shutdown_core()
    call mpi_finalize(ierror)

    call date_and_time(tdate, ttime, tzone, tvalues)
    end_time = tvalues(5)*60*60 + tvalues(6)*60 + tvalues(7)
    print *, "Completed in ", (end_time - start_time), "seconds"

    return
end program
```

Listing 6.1: Example of a user's main simulation program. This example is capable of running on any amount of CPU cores.

operation and that `mpi_finalize` should be called before terminating.

The parallelisation was implemented using MPI which is a message passing interface standard. The *Open MPI* [9] implementation of this standard was used. MPI allows operations such as the broadcasting, scattering and gathering of data to be performed.

A simple comparison in performance between a simulation using only a single processor and one using two processors is made in table 6.1. On average, the simulation running two processes complete in about 67% of the time of the single process simulation. The only difference from the user's perspective is the command used to execute the simulation. A single process simulation can be started by simply running the executable, but in order to start a simulation with two processes, a user must execute `mpirun -np 2 mysimulation`, where `mysimulation` refers to the executable file. Using this command, two copies of `mysimulation` is started, their ranks are assigned and communication between the two processes can take place.

Table 6.1: A comparison in execution times of simulations using one and two processors respectively

Test	1	2	3	4	5
One processor [seconds]	42	80	11	1025	50
Two processors [seconds]	25	52	7	805	33

Combined with the improved use of compiler optimisation discussed in section 3.3 and the modification of the initial estimate of the vector potential discussed in section 3.6, it is realistic to say that on a computer with a dual core processor, simulation times can be reduced to 30% of the time it would have taken using the original package. This improvement is expected to be greater on computers with more CPU cores available.

6.4.3 Parallel optimisation capabilities

The development of parallel computing capabilities was, however, undertaken with cluster computers as the targeted platform. The idea was to drastically reduce the time needed to perform optimisations by throwing much more computing power at the problem, using the scheme depicted in figure 6.1. In order to do this, it was necessary to implement the parallel capabilities discussed in section 6.4.2.

During the course of this work, a library was developed that would allow a user to setup a *VisualDOC* optimisation easily and perform this optimisation on a cluster using MPI. The library is capable of distributing design points intelligently among the available CPU cores, a task which is not simple because the amount of design points that can be simulated simultaneously vary during the course of the optimisation. This parallelisation is illustrated in figure 6.1 by the optimiser requesting many input parameter sets to be evaluated simultaneously.

Apart from the broadcasting, scattering and gathering operations mentioned in the previous section, the optimisation library also makes use of MPI's group and communicator management functionality. This allows new groups or *communicators* to be created dynamically. A *communicator* is an object used to identify the processes taking part in data communication operations. Data communication operations, such as broadcasting can then be performed across all available processes or only across the members of specific *communicators*. This functionality is vitally important in the optimisation library.

Consider for example an analysis with 10 design variables, consisting of 40 time-steps, running with 40 available CPU cores[1]. When a gradient-based optimisation is performed, the amount of design points that can be simulated simultaneously will typically be 10 when gradients are calculated, or 1 otherwise. When only one point is available, it is best to throw all 40 CPU cores at this one point. When 10 points are available, it is best to assign 4 CPU cores to each of the design points, thus reducing overheads. The developed library is capable of performing this type of dynamic allocation of CPU cores. In turn, *SEMFEM* only requires an MPI communicator object, which the library provides, to know how many nodes it has available and how to distribute the time-steps to the CPU cores assigned to the specific design point simulation.

Although successful optimisations have been run on a cluster and the dynamic allocation of CPU cores functions well, the expected performance benefits have not yet been realised. The reason for this is that problems were experienced with the cluster that was used for testing. The amount of CPU cores that could be used was limited because there was a problem with the integration of the MPI implementation and the resource allocator used on the cluster. Towards the end of this work, this issue was partly resolved, but limited time remained for testing.

6.5 Conclusions

The current trend in high performance computing is to make more and more use of parallel processing. During the course of this work, a good foundation has been laid to allow *SEMFEM* to exploit the parallel processing capabilities of modern computers. Improvements in simulation times on single computers with multiple CPU cores have already been realised. Although optimisation on computer clusters have not delivered the expected performance benefits yet, this area of application remains promising.

[1]The author had access to a 168 core cluster at the University of Stellenbosch. The portrayed scenario is realistic.

Part II

Application

Chapter 7

Case studies

7.1 Introduction

Having discussed the mathematics and programming behind the *SEMFEM* package, this chapter serves to illustrate the use of *SEMFEM* and to verify the accuracy of the calculations presented in part I of this thesis. Several case studies will be presented, each serving to illustrate a specific use or to verify the accuracy – both in derivation and implementation – of particular calculations.

The accuracy of *SEMFEM* is validated by comparison with two commercial finite element packages, *Ansoft Maxwell 12* and *Magnet 6* from the *Infolytica Corporation.* This required that the same model be simulated in *SEMFEM* and the package used as a benchmark. When these models were defined, every effort was made to ensure that the models were equivalent, including exact dimensions and material characteristics. For example, even though both *Maxwell* and *Magnet* provided a material, *1010 steel*, different B-H curves where used in *SEMFEM* for *Maxwell's* and *Magnet's* version of this material. Care was also taken to ensure that the characteristics of magnets were exactly equivalent between models. High grade Neodymium Iron Boron magnets (typically grade N48) were used in the simulations to follow.

Note that the drawings of the machines do not necessarily represent the dimensions of the actual simulated machine. The purpose of the drawings is to illustrate the topology. Where dimensions are shown, it is merely to give an indication of the size of the simulated machine.

7.2 Case study: Rotating machine

7.2.1 Description

The first case study to be considered is a simulation of the radial flux air-cored permanent magnet machine presented by Stegmann [25]. A one eighth model of this machine is shown in figure 7.1. The machine has an outer diameter of about 500 mm and an active axial length of 76 mm. This case study is used to validate the accuracy of the previously unimplemented double air-gap functionality in rotating machines, as well as the accuracy of the new magnet modelling scheme discussed in section 3.5. The commercial finite element analysis package, *Ansoft Maxwell 12*, was used as a benchmark.

7.2.2 Simulation results

The torque on the rotor of the machine and the flux linkage of the three phase winding were used to compare the simulations. These quantities were chosen because they were considered to be the most important results from the simulation, since many other machine characteristics can be derived from them. The torque calculation is also highly sensitive to the field in the air-gap and is thus a good metric to use for comparisons. The results from two different scenarios were compared, the one employing a finer mesh than the other. Comparisons of the torque waveforms for the two scenarios are shown in figures 7.2 and 7.3. In both cases, the results are in good agreement. The exact cause of the small offset in the torque waveforms (about 0.23%) is not known. This discrepancy was considered to be insignificant. Figure 7.4 shows a comparison of the flux linkages. In this

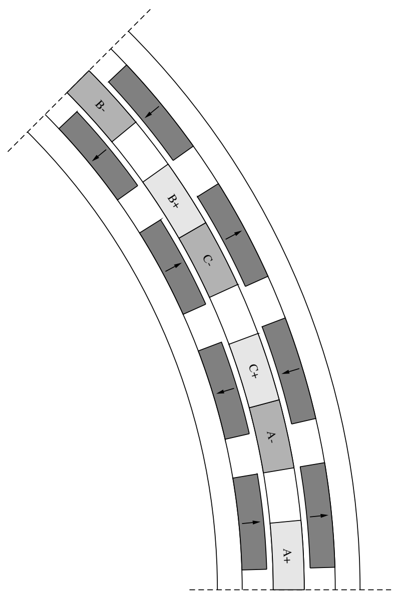


Figure 7.1: A radial flux, air-cored permanent magnet machine.

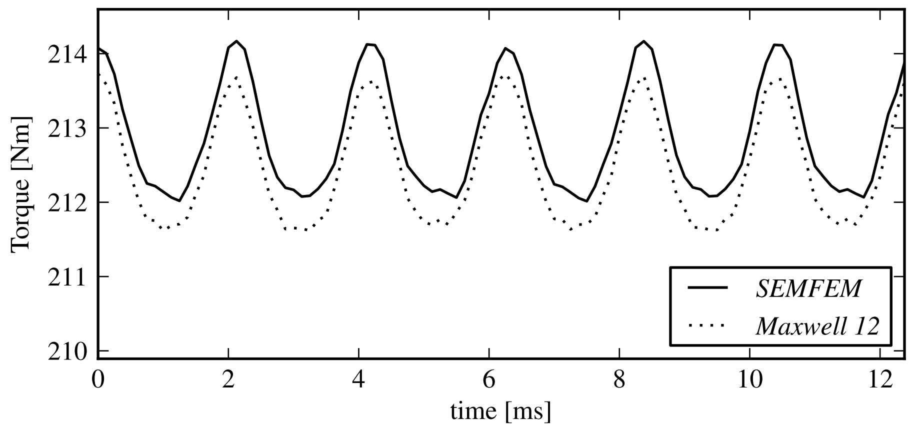


Figure 7.2: Comparison of the torque waveforms of the machine shown in figure 7.1 from *SEMFEM* and *Maxwell 12* using a fine mesh.

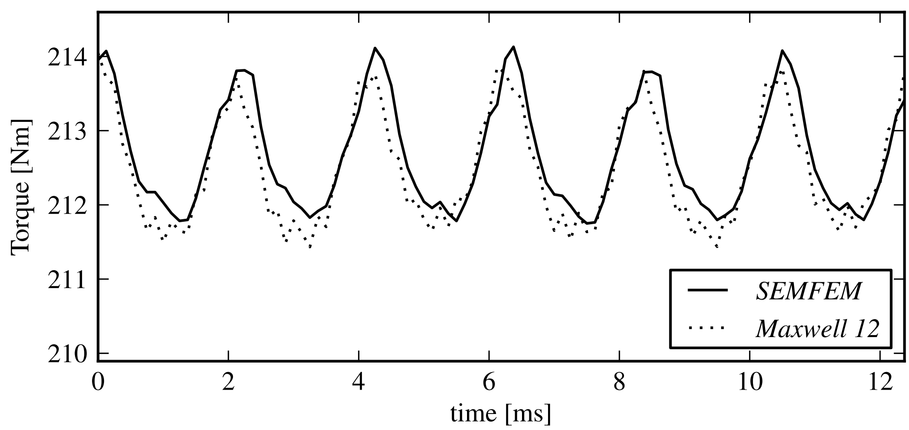


Figure 7.3: Comparison of the torque waveforms of the machine shown in figure 7.1 from *SEMFEM* and *Maxwell 12* using a coarse mesh.

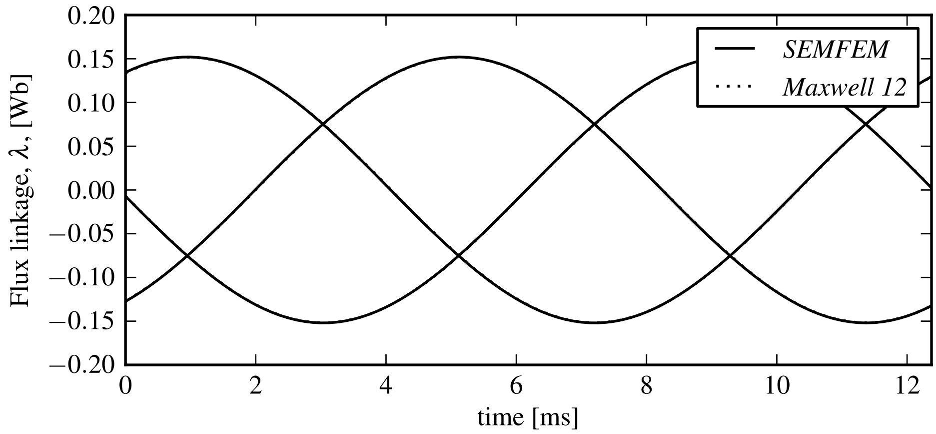


Figure 7.4: Comparison of flux linkage waveforms from *SEMFEM* and *Maxwell 12*.

case only one figure is shown because the results, which are almost identical, do not differ noticeably for the two scenarios.

The simulation of this machine was also presented by Gerber et. al. [10] where the computational time of two *SEMFEM* simulations, the ones considered in this section and the next, were compared with that of commercial finite element packages. The results showed that *SEMFEM* can deliver similar performance to the commercial packages. The comparison of computational times for this simulation is shown in table 7.1. Note that even the simulation with a coarse mesh was still a relatively large problem because a fine mesh was required to accurately calculate the torque ripple. For smaller problems, *SEMFEM* outperformed the commercial packages by an increasing margin.

After implementing the magnet modelling scheme of section 3.5, a comparison was also made between the two magnet modelling techniques. Once again, the flux linkages are identical and will not be shown here. The torque waveforms are compared in figure 7.5. This result proves that the two methods, as implemented in *SEMFEM*, are not only equivalent in theory but also as far as practical results are considered.

Table 7.1: Performance comparison of finite element simulation packages.

	Maxwell 12 [seconds]	**SEMFEM** [seconds]
32-bit Windows XP (fine mesh)	684.6	635.4
64-bit Linux (fine mesh)	not available	495.5
32-bit Windows XP (coarse mesh)	333.3	245.1
64-bit Linux (coarse mesh)	not available	174.1

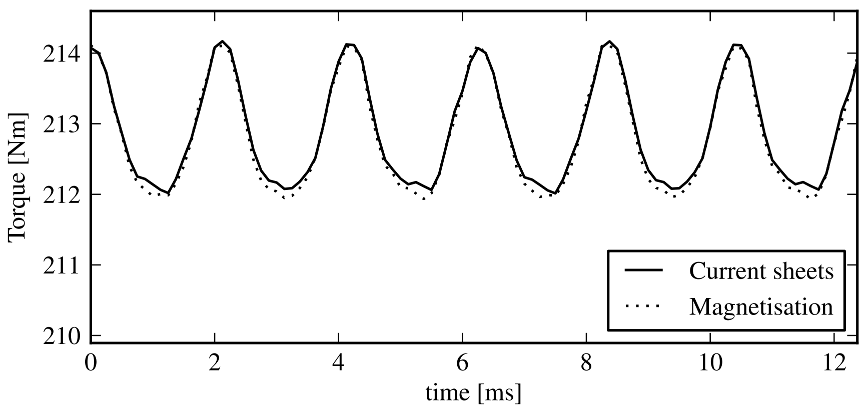


Figure 7.5: Comparison of torque waveforms from *SEMFEM* using the current sheet and the extra magnetisation term approaches.

7.3 Case study: Flat linear machine

7.3.1 Description

The simulation of the flat short stroke linear machine shown in figure 7.6 is presented as an example of a class II problem. Although it is possible to model only half of the machine shown here by applying a Neumann boundary condition on the dashed line through the centre of the model in figure 7.6, the full machine was also simulated to test the implementation of multiple air-gaps for flat linear machines. Although not shown here, the results from the simulations of the full model and the half model are in very good agreement. In this case study, *SEMFEM* simulations were compared against the commercial package, *Magnet 6*.

7.3.2 Simulation results

As was done in the previous section, the force and flux linkage waveforms are used for comparison. Figures 7.7 and 7.8 show the comparisons of the flux linkage and force waveforms for this machine. The results are in very good agreement.

As mentioned in section 7.2, this simulation was the second to be considered in [10]. The measured computational times for this simulation are reported in table 7.2. In this case, *SEMFEM* outperformed the commercial package by a substantial margin.

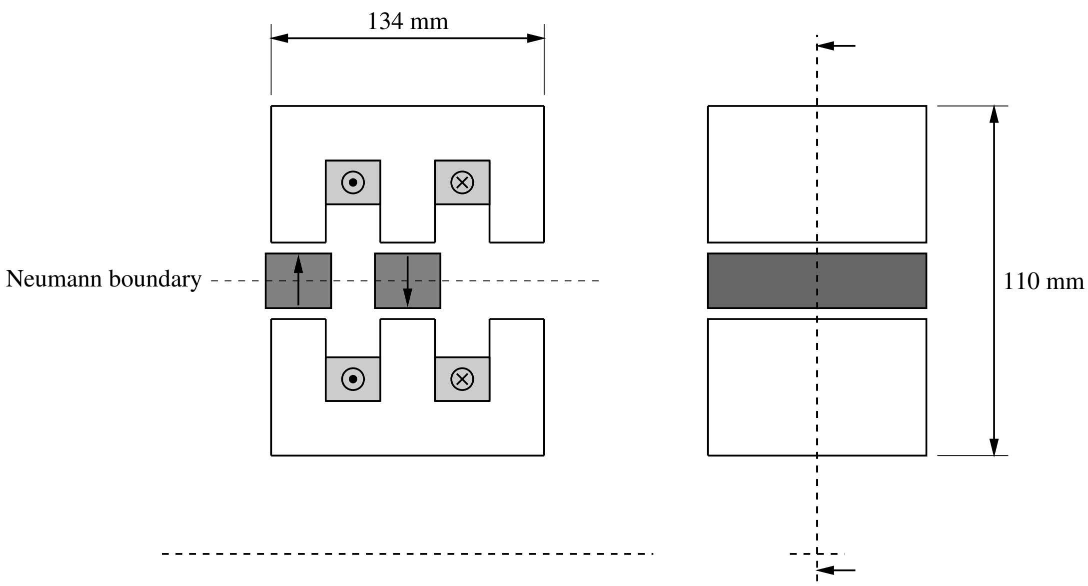


Figure 7.6: A flat short stroke linear machine.

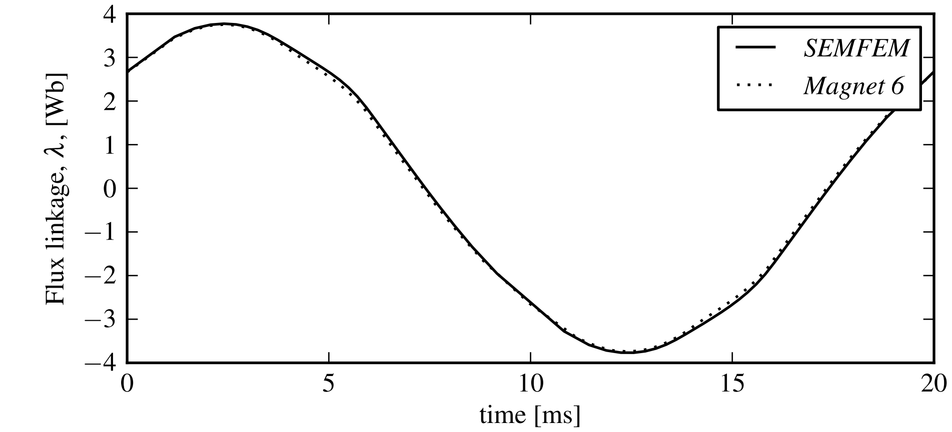


Figure 7.7: Comparison of the flux linkage waveforms from *SEMFEM* and *Magnet 6*.

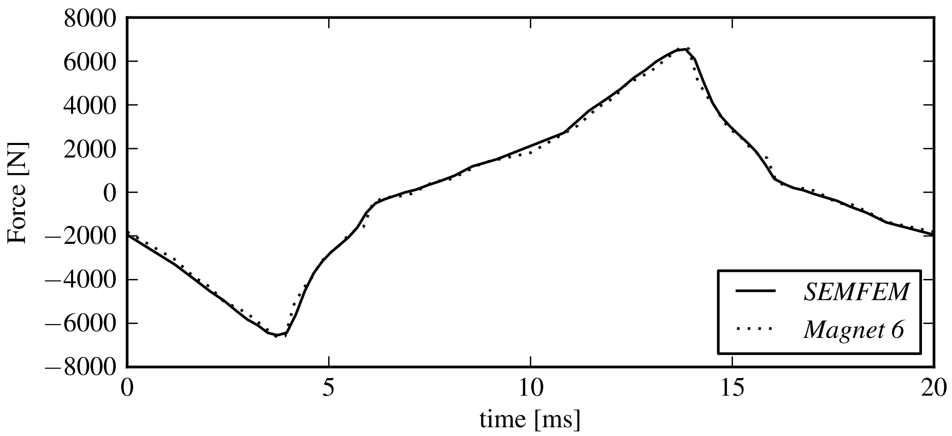


Figure 7.8: Comparison of the force waveforms from *SEMFEM* and *Magnet 6*.

Table 7.2: Performance comparison of finite element simulation packages.

	Magnet 6 [seconds]	**SEMFEM** [seconds]
32-bit Windows XP	42.1	21.0
64-bit Linux	not available	12.7

7.4 Case study: Tubular linear machine

7.4.1 Description

The case study presented here serves to verify the accuracy of all the derivations for axisymmetric problems presented chapter 5, as well as the implementation thereof. The machine used is a tubular version of the machine used in the previous case study and is shown in figure 7.9.

7.4.2 Simulation results

Once again, the accuracy of *SEMFEM* was verified by comparing simulations with the commercial finite analysis program, *Magnet 6*.

Three test cases were created by using the same cross sectional dimensions and varying the radius of the innermost part of the machine, r_i in figure 7.9. This was done because the solution of the field in the air-gap varies with r and it was necessary to verify that the solution is accurate over all r. The values of r_i for the different test cases are listed in table 7.3. Figures 7.10 to 7.15 show the comparisons of the force on the translator and flux linkage waveforms for the three test cases. Note that the force on the translator is directly derived from the solution of the field in the axisymmetric air-gap element. Therefore, the comparison of the force is especially important to validate the derivation and implementation of the axisymmetric air-gap element.

The results for the first two cases, are in very good agreement.

For the third case, it is noticed that the torque waveform produced by *SEMFEM* fails to capture the higher frequency components of the torque. This is evident from the fact that the torque waveform produced by *SEMFEM* is a slightly smoothed version of that produced by *Magnet 6*. The lack of high frequency information in the torque waveform produced by *SEMFEM* is a result of the limitation on the number of Fourier terms in the stiffness matrix calculation of the axisymmetric air-gap element. This issue was discussed at the end of section 5.3.2. Note, however, that this discrepancy only occurred at a radius of 1 meter. In the short term future, *SEMFEM* will probably not be used to model such large axisymmetric machines. In any case, the solution to this problem, as recommended in section 5.3.2, should be implemented.

Table 7.3: Values of r_i for three test cases.

Test case	1	2	3
r_i [mm]	10	100	1000

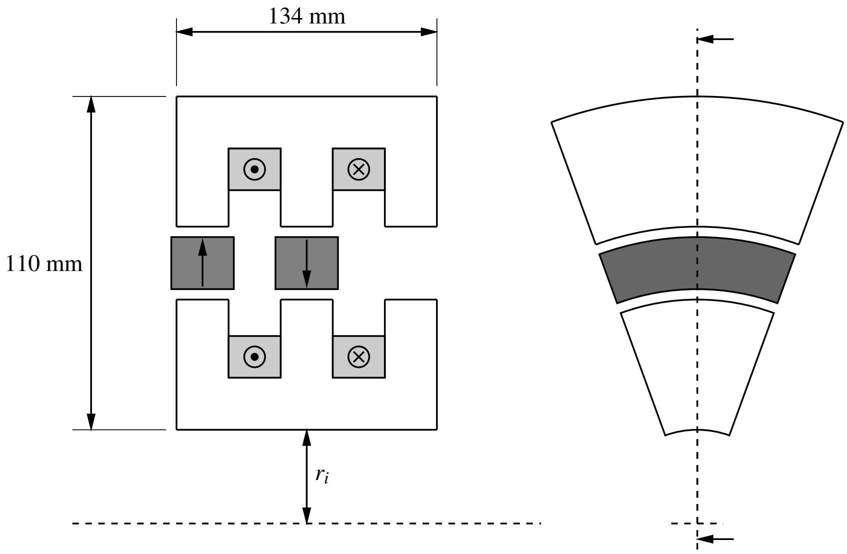


Figure 7.9: An example of a linear axisymmetric machine.

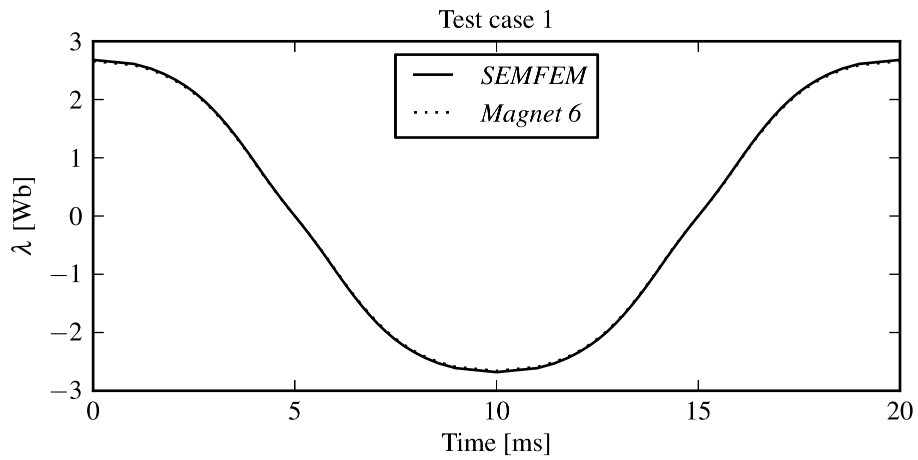


Figure 7.10: Comparison of the flux linkage waveforms from *SEMFEM* and *Magnet 6* for test case 1.

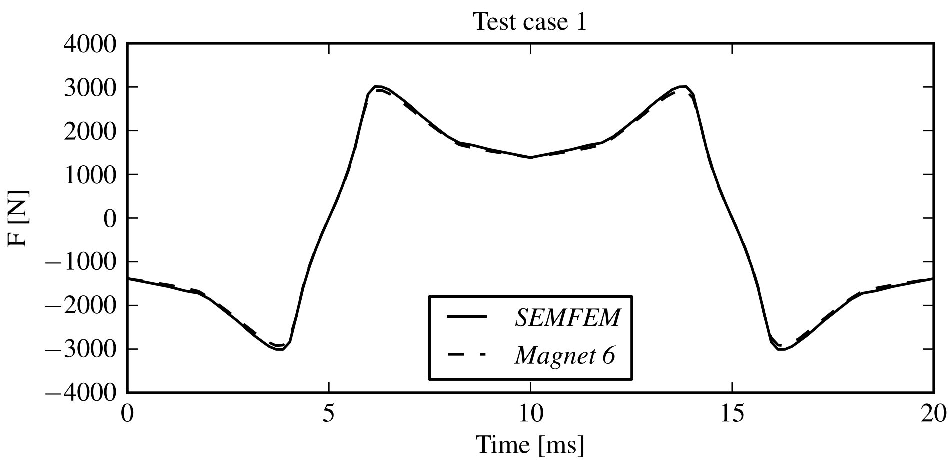


Figure 7.11: Comparison of the force waveforms from *SEMFEM* and *Magnet 6* for test case 1.

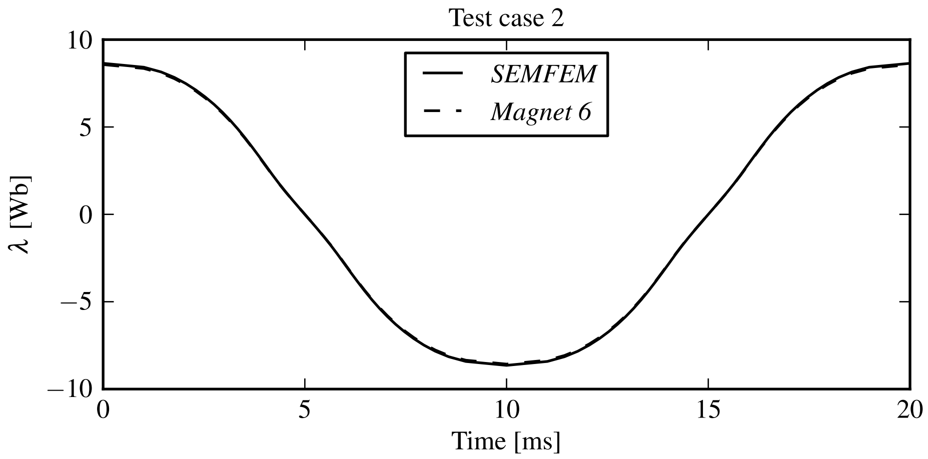


Figure 7.12: Comparison of the flux linkage waveforms from *SEMFEM* and *Magnet 6* for test case 2.

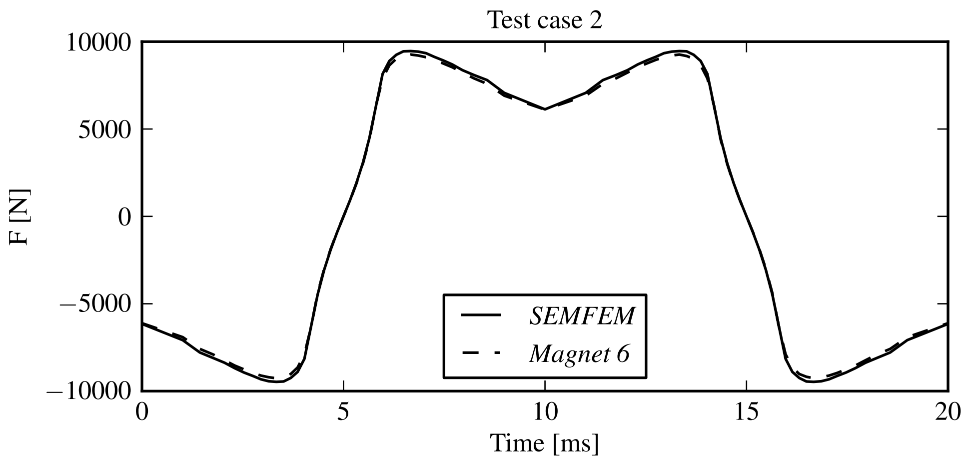


Figure 7.13: Comparison of the force waveforms from *SEMFEM* and *Magnet 6* for test case 2.

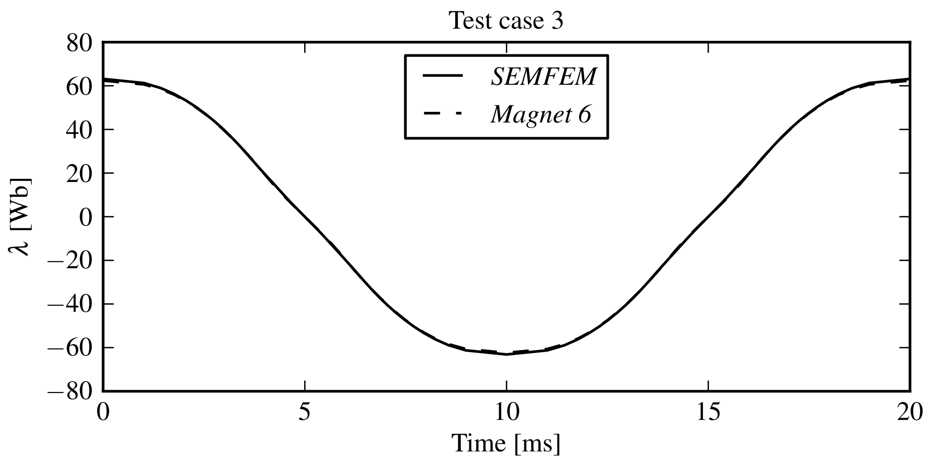


Figure 7.14: Comparison of the flux linkage waveforms from *SEMFEM* and *Magnet 6* for test case 3.

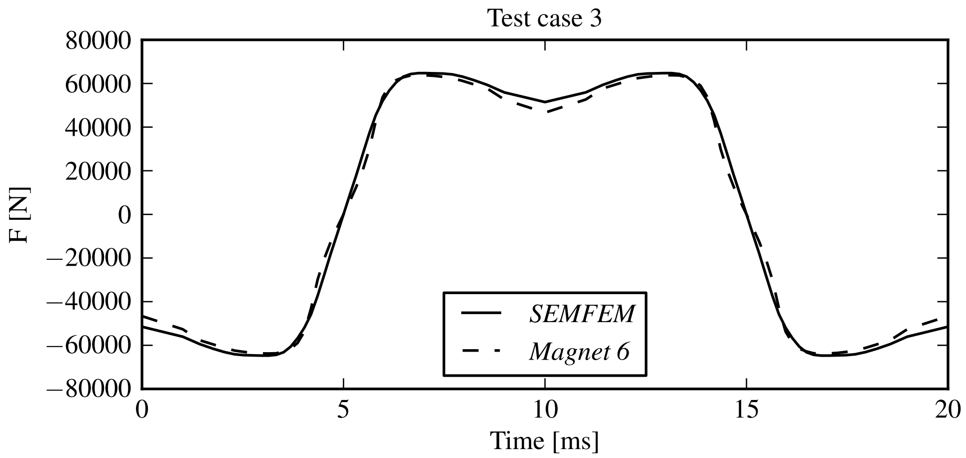


Figure 7.15: Comparison of the force waveforms from *SEMFEM* and *Magnet 6* for test case 3.

7.4.3 Conclusions

The accuracy of all the calculations of chapter 5, including the calculation of the local stiffness matrix coefficients for triangles and for axisymmetric air-gap elements, the calculation of the coefficients of the Jacobian matrix and the calculation of force and flux linkage, have been verified.

7.5 Case study: Electrical machine with integrated magnetic gear

This case is study is presented simply to illustrate a case where more than two air-gaps are required. It also employs the new magnet modelling technique discussed in section 3.5. The accuracy of this simulation was not verified.

Figure 7.16 shows a finite element model of an integrated magnetic gear and electrical machine. Note that the machine has three air-gaps. The outer part is the low speed rotor, then follows a stationary ring of iron pieces, followed by the high speed rotor and stator at the inner part of the machine. The stator winding is not shown. The configuration, as it is shown here, is not necessarily a practical device. The purpose is only to illustrate that *SEMFEM* can simulate machines such as this. A flux density colour-map of the machine is shown in figure 7.17.

The reason that the entire machine is modelled and not only a slice, is that there is no symmetry that can be exploited. This further illustrates the modelling capabilities of *SEMFEM*.

In figure 7.16, there are no current sheets because the magnet modelling technique of section 3.5 was used. It is noted that the model, as shown in figure 7.16, has significantly fewer elements than the equivalent model using the current sheet approach would have had because extra small elements are required for the current sheets of each magnet. This means that the stiffness matrix of the model of figure 7.16 will be smaller and the solution can be obtained faster. The extent of this improvement was however, not measured.

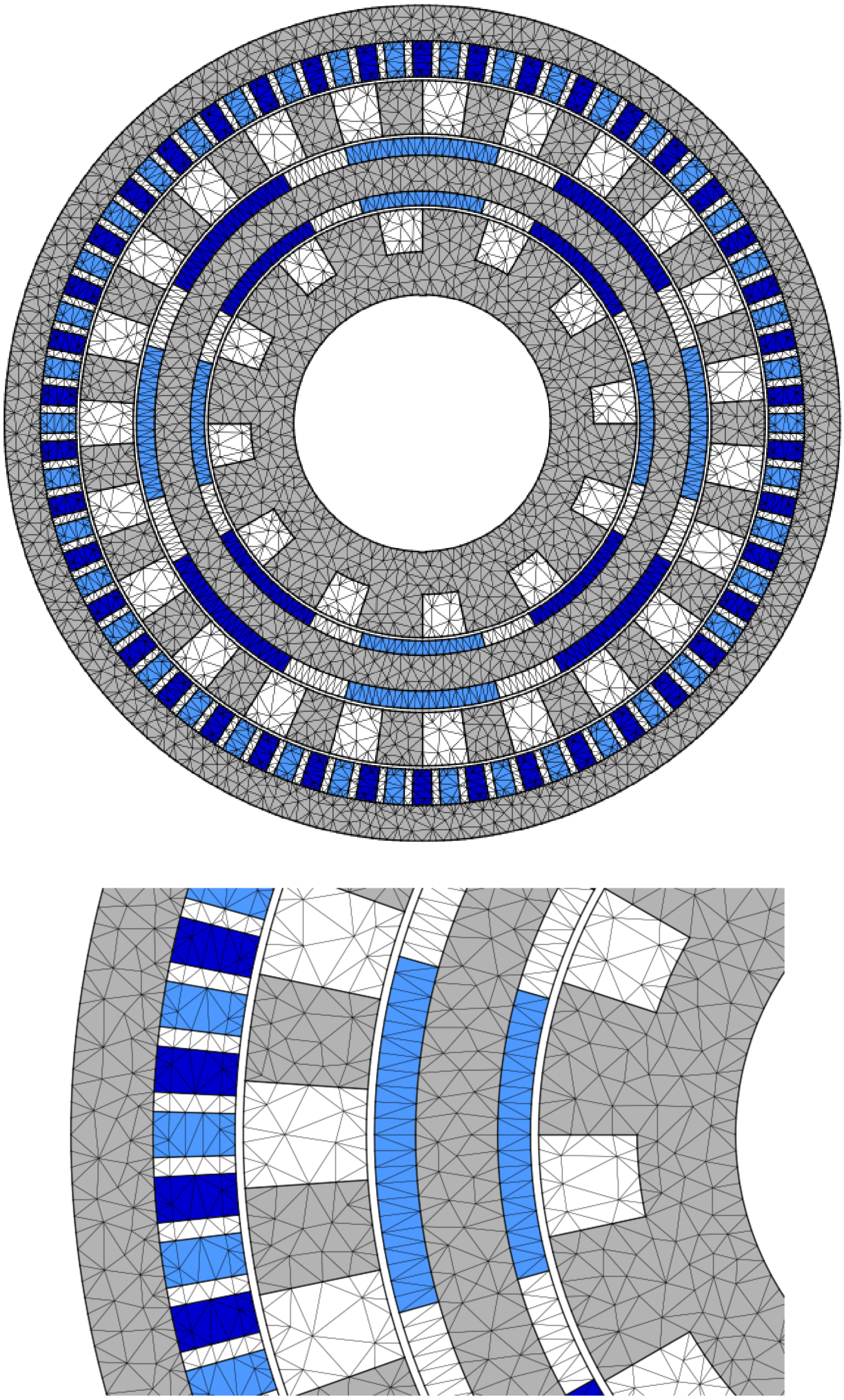


Figure 7.16: An integrated magnetic gear and electrical machine (stator winding not shown).

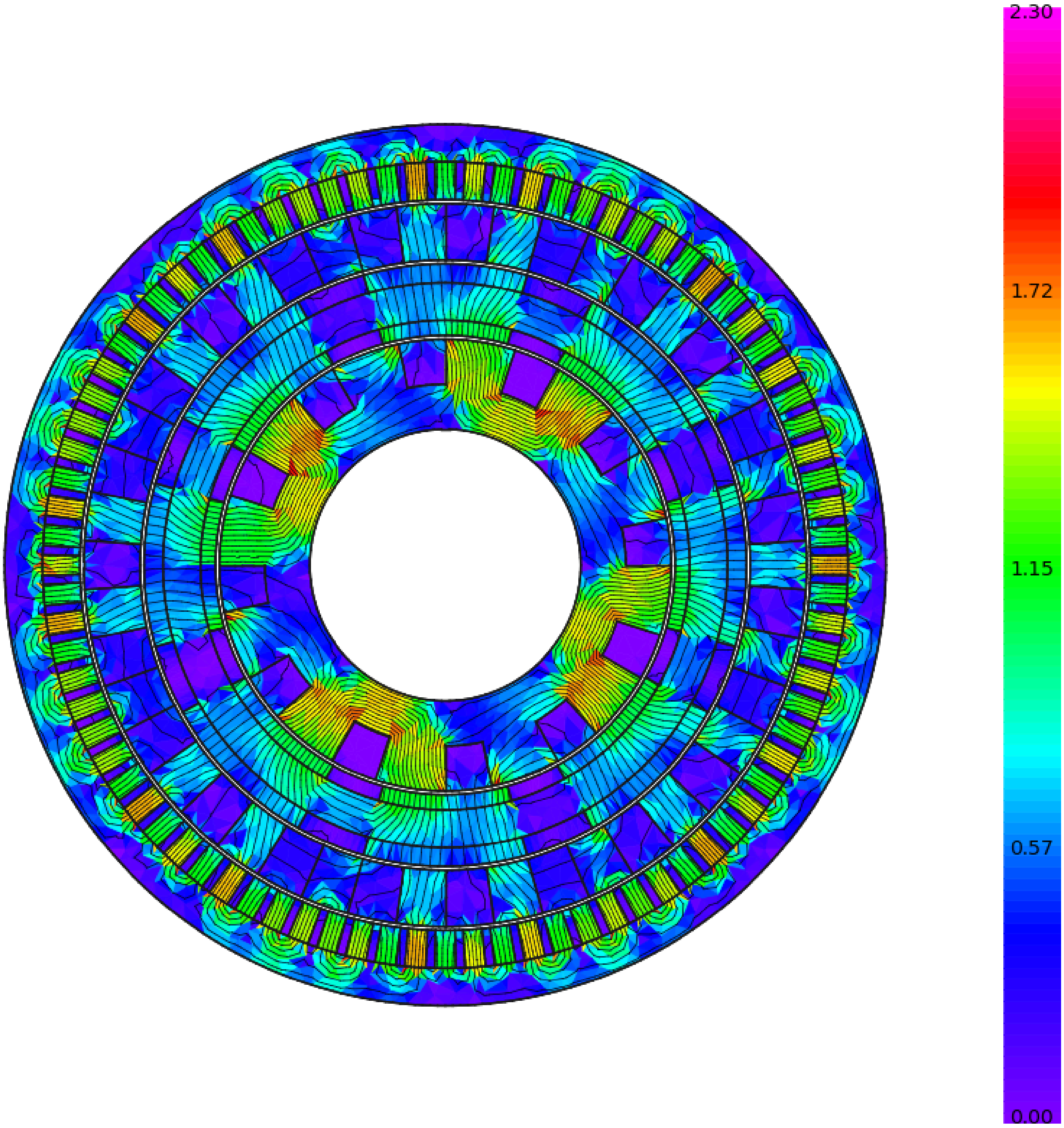


Figure 7.17: Flux density colour-map of a magnetic gear.

7.6 Case study: Optimisation

7.6.1 Description

In this case study, the use of *SEMFEM* to optimise a machine is considered. In the work presented here, no great effort was made to ensure that the final design is really the optimal design. The purpose is rather to illustrate how the optimisation process works. The optimisations presented here were run on a single computer with a dual core processor.

The tubular linear machine shown in figure 7.18 is used as an example. The cross section is rotated about the axis shown. The coil rings in the centre of the configuration form the translator. It is therefore a moving coil type linear machine. All the dimensions shown are the design variables of the optimisation problem. The optimisation problem is formulated

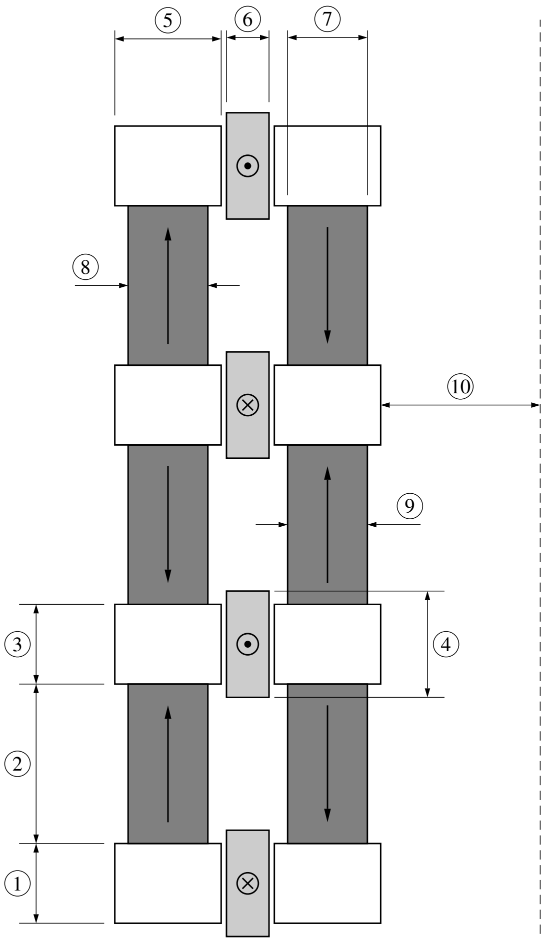


Figure 7.18: A linear air-cored machine for a Stirling engine application.

as follows:

$$\begin{aligned}\text{Minimise}: &\quad \text{Total mass}\\ \text{Subject to}: &\quad \text{Output power} > 3 \text{ kW} \qquad (7.1)\\ &\quad \text{Translator mass} < 3 \text{ kg}\end{aligned}$$

$$X_{li} <= X_i <= X_{ui} \qquad (7.2)$$

where the total mass, the output power and the translator mass are all functions of the set of design variables $\mathbf{X}$ and the search space is defined by the lower and upper bounds X_{li} and X_{ui}. For example, for dimension 2 in figure 7.18, the lower and upper bounds were defined by $X_{li} = 10$ mm and $X_{ui} = 50$ mm, while the initial value was set at $X_i = 20$ mm. The goal of this optimisation is to design a machine that can be applied in a 3 kW Stirling engine application with the crucial requirement that the translator mass may not exceed 1 kg/kW.

7.6.2 Optimisation results

Once the optimisation problem was properly defined, a couple of initial optimisations were run to ensure that the output produced by *SEMFEM* is realistic and that the simulation does not fail because of infeasible sets of design variables. Experience has shown that this is necessary because it often happens that as the optimiser varies the values of the design variables, an unforeseen situation arises where, for example different components overlap and the mesh cannot be constructed properly. The solution is usually to modify the bounds on the search space or to redefine some of the design variables as fractions of other design variables. This situation is not unique to optimisation using *SEMFEM*.

Once a reliable analysis was set up, the accuracy of the simulation was increased by using a finer mesh and three of the methods available in *VisualDOC* were used to optimise the machine so that results can be compared. The three methods used were the modified method of feasible directions (MMFD), sequential linear programming (SLP) and sequential quadratic programming (SQP). The results from the three optimisations as well as the relevant values of the initial design are given in table 7.4. Note that the initial design did not satisfy the constraint on the translator mass and that the algorithms from *VisualDOC* are capable of handling this and produced results that satisfy all the constraints in all three cases. Although, it appears that the constraint on output power was not satisfied, *VisualDOC* allows small violations of constraints. If it is required, the criteria for satisfying the constraints can be enforced more strictly. The number of analyses required to reach the optimal design is also shown because it is an important measure of the efficiency of the optimisation process.

Table 7.4: Initial design and final designs from different optimisation methods.

Method	Initial design	MMFD	SLP	SQP
Total mass [kg]	73.3	57.2	46.6	29.5
Output power [W]	3448	2991	2992	2993
Translator mass [kg]	5.4	3.0	3.0	3.0
Number of analyses	n.a.	246	473	318

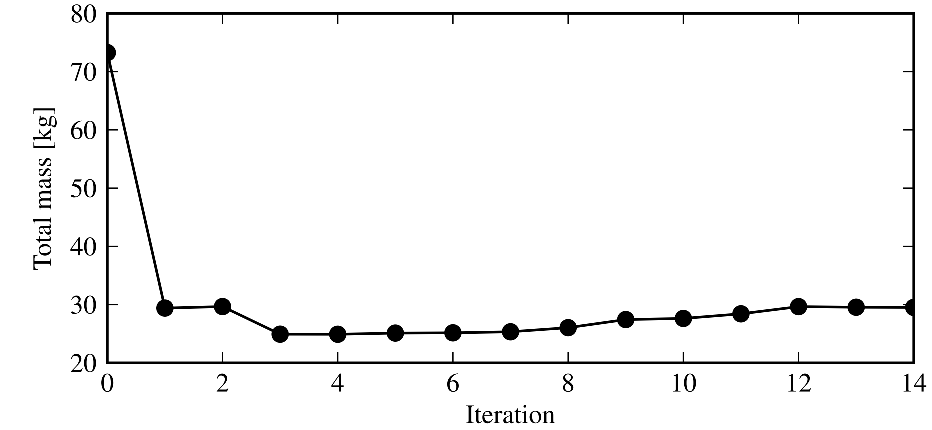
In this case, the SQP method produced the best result by a substantial margin. It may be possible to achieve better results with the other methods too by tuning some of the optimisation algorithm's parameters such as the size of the finite difference steps used to calculate gradients. In general, it is also good practice to run optimisations with different initial designs. If the same optimum is reached, one can be more confident that this design is indeed the optimal design. As mentioned earlier however, the purpose here is just to illustrate how *SEMFEM* is used to optimise a machine.

Figure 7.19 illustrates the progress of the SQP optimisation process. Note that although the value of the objective function did not vary significantly between iteration 1 and 14, the constraint on the output power was only met at iteration 12.

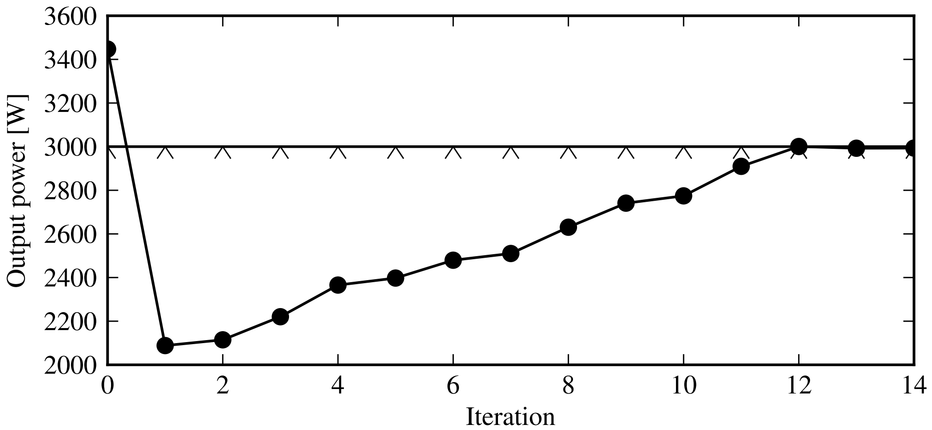
To give an indication of how the dimensions were varied by the optimisation process, the finite element models for the initial and final designs are shown in figures 7.20 and 7.21 respectively. What is not clear from the figures is that the initial design had an inside radius of 250 mm and an outside radius of 297 mm, while in the final design, the inside radius was 50 mm and the outside radius 106 mm. A flux density colour map of the final design is also shown in figure 7.22.

7.6.3 Conclusions

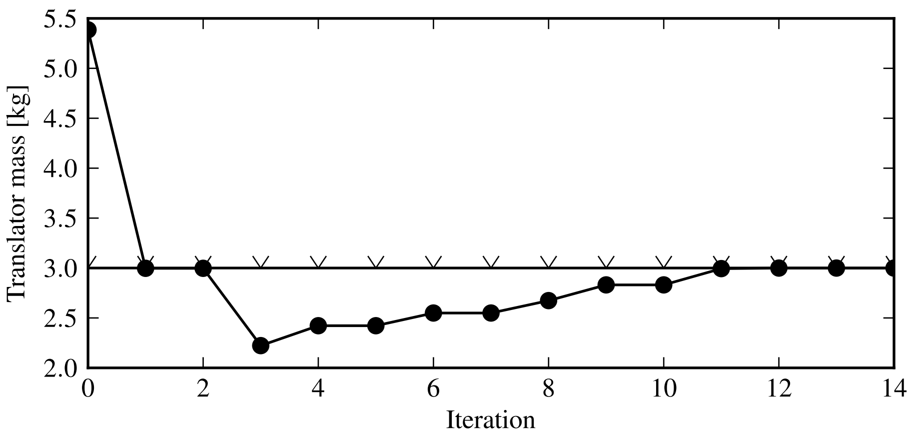
The use of *SEMFEM* in the optimisation of a machine has been illustrated. Although the full power of the optimisation library was not used in this case study, it did enable the use of both available cores. When a single analysis point was requested, both cores where used to perform the analysis. When gradients were calculated, the total points to be simulated were divided between the two cores. The optimisation library also provides a simple interface to the user, hiding many of the complexities involved in setting up the problem.



(a) Objective function (Total mass).



(b) Constraint 1 (Output power).



(c) Constraint 2 (Translator mass).

Figure 7.19: Results from the sequential quadratic programming optimisation.

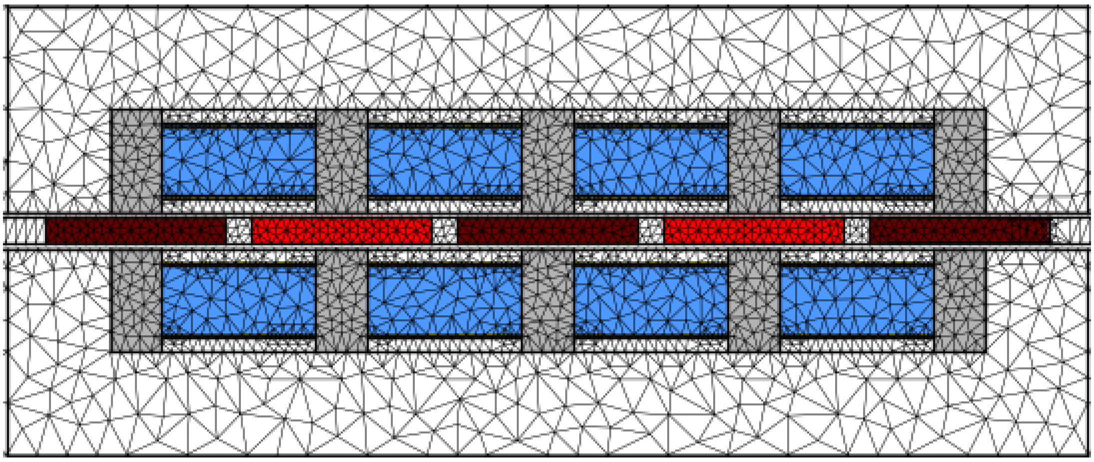


Figure 7.20: Finite element model of the initial design.

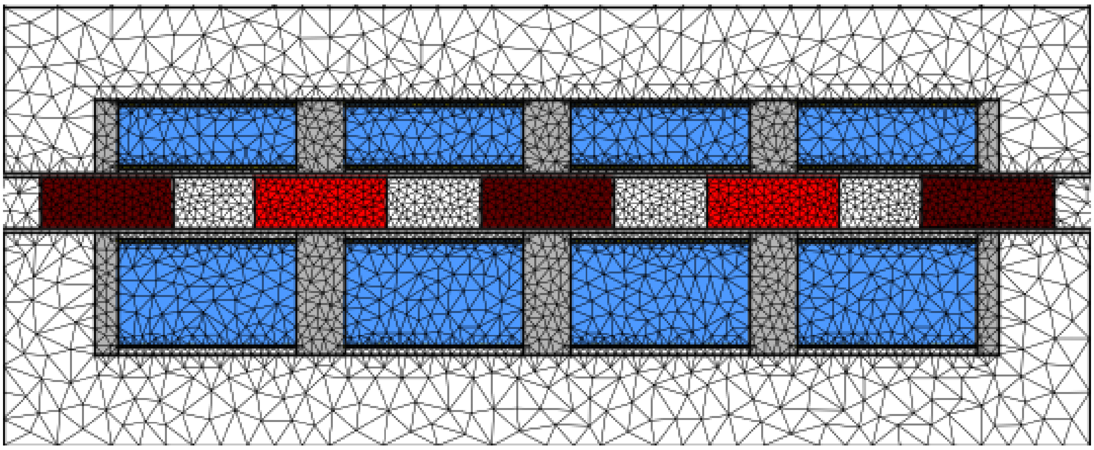


Figure 7.21: Finite element model of the final design.

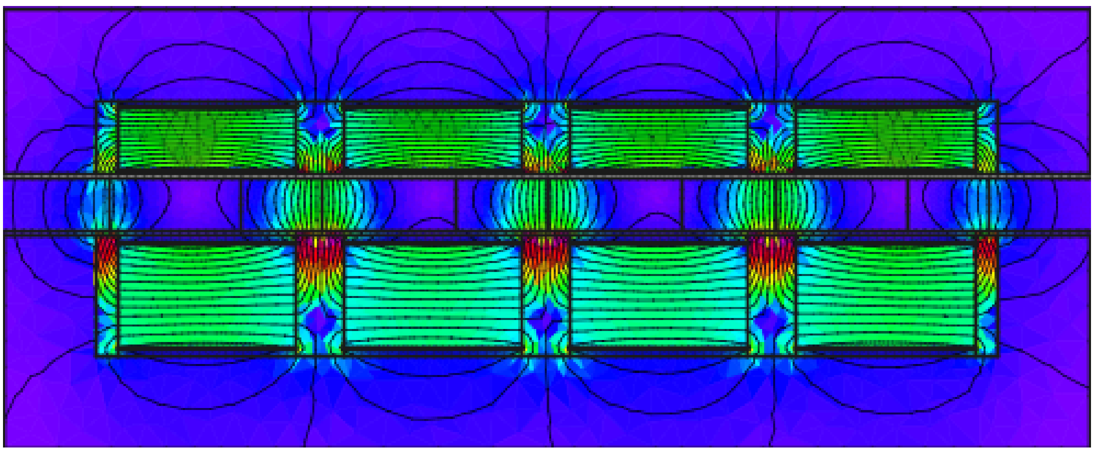


Figure 7.22: Flux density colour map of the final design.

Chapter 8

Conclusions and Recommendations

8.1 Accomplished goals

The goals set out in section 1.4.2 can be considered accomplished based on the following:

- It was illustrated in section 7.4 that the *Cambridge* package was successfully extended to allow the simulation of class III (linear axisymmetric) problems. This part of the work included the derivation of the axisymmetric air-gap element, which, along with the original air-gap element derived by Razek et. al. [2] and the extension of the method to the Cartesian coordinate system by Wang et. al. [29, 30], completes the derivation of all two-dimensional air-gap elements.
- The ability to model machines with multiple air-gaps, using the air-gap element method, was successfully implemented in *SEMFEM*. This functionality was demonstrated in all the case studies presented in chapter 7.
- The user's experience of the package has been vastly improved by the new mesh generation capabilities and the creation of the *SEMFEM* core library. To give an indication of how much simpler it is for a user to use the new program, table 8.1 gives an indication of the size of different components of the *SEMFEM* package and a couple of user simulations. The size is measured in the amount of lines of source code. In the past the user had to manage everything. With the restructured package, all the functionality of the core and the optimisation library becomes available by simply linking against the libraries. In the original program received by the author, the "user code" needed to draw the mesh (the `ee_pol` subroutine described in section 2.2.6) for the specific machine was about 1300 lines long and offered very limited support for varying the mesh density easily. Referring to the last case study in table 8.1, the user code needed to define an entire optimisation process was 1200

Table 8.1: Size of different components measured in lines of source code.

Component	Lines of source code
SEMFEM	**32 500**
Core (excluding Triangle)	15 500
Triangle	16 300
Optimisation library	700
User simulations	
Case study: Integrated magnetic gear	400
Case study: Rotating machine	800
Case study: Tubular linear machine	900
Case study: Optimisation	1200

lines long. This includes the code to define design variables, the objective function and constraints, initialize the core, draw the mesh, run the simulation and do post-processing. Furthermore, complete control over the mesh density in various parts of the machine can be exercised by simply adjusting a few mesh density parameters.

- A powerful platform for the optimisation of electrical machines has been established by combining the *VisualDOC* suite and *SEMFEM*. Its use was demonstrated in section 7.6. The use of parallel processors was introduced at the level of the simulation, where performance benefits have been realised. Parallelisation was also introduced at the level of optimisation, where the expected benefits have not yet been realised. This area, however, remains promising. The developed optimisation library that now forms part of the *SEMFEM* package can probably even be used with other finite element packages performing the analyses, although this was not its intended use.

8.2 Recommendations regarding future development

Apart from the less significant recommendations made throughout this thesis, the following recommendations regarding future development of *SEMFEM* are made:

SEMFEM Python binding It is often desirable to use Python as a post-processing engine, but the current implementation of this functionality requires a complex piece of "glue code" to allow calling Python functions from the Fortran simulation program. It may be a better idea to provide a full Python language binding for *SEMFEM* using the *f2py* tool [20]. Because of the ease with which *f2py* can wrap Fortran code, this may be an almost straight forward task.

Time-harmonic solver In order to allow the modelling of induction machines, time-harmonic simulation capabilities are required. This functionality can be added to *SEMFEM*.

3D Problems *SEMFEM* can be extended to allow the simulation of three-dimensional problems. The most difficult challenge to overcome in this area may be the problem of generating quality three-dimensional meshes. Unfortunately, *Triangle* is not capable of this at present.

Loss calculations Modelling of eddy-current and core losses is currently not possible using *SEMFEM*. Investigation of the possibilities in these areas is required.

Movement handling Although the air-gap element approach of dealing with movement between different parts of the mesh currently works well in *SEMFEM*, it is still recommended that the use of other techniques are investigated. The main reasons for this recommendation are that other techniques may offer superior performance in terms of computational time and that if *SEMFEM* is to be extended for three-dimensional problems, another technique of handling movement will be required.

Graphical output Although the Python field plot viewer discussed in section 3.7 is a very capable application, it is unfortunately, rather slow. It is recommended that this application is rewritten, using a higher performance, cross-platform graphical library. The development of a full graphical user interface for *SEMFEM* is absolutely not recommended. In the author's experience, even commercial packages are best used through their scripting interfaces, especially for optimisation purposes.

Documentation A good tutorial on the use of *SEMFEM* and proper documentation of the interfaces provided to users are required.

References

[1] A.A. Abdel-Razek, J.L. Coulomb, M. Feliachi, and J.C. Sabonnadiere. The calculation of electromagnetic torque in saturated electric machines within combined numerical and analytical solutions of the field equations. *IEEE Trans. Magn.*, 17(6):3250–3252, November 1981.

[2] A.A. Abdel-Razek, J.L. Coulomb, M. Feliachi, and J.C. Sabonnadiere. Conception of an air-gap element for the dynamic analysis of the electromagnetic field in electric machines. *IEEE Trans. Magn.*, 18(2):655–659, March 1982.

[3] K.J. Binns, P.J. Lawrenson, and C.W. Trowbridge. *The Analytical and Numerical Solution of Electric and Magnetic Fields.* John Wiley & Sons Ltd., Buffins Lane, Chichester, West Sussex PO191UD, England, 1992.

[4] M.V.K. Chari and P. Silvester. Finite-element analysis of magnetically saturated d-c machines. *Power Apparatus and Systems, IEEE Transactions on*, PAS-90(5):2362 –2372, sept. 1971.

[5] R.W. Clough. The finite element method in plane stress analysis. *Proceedings of 2nd ASCE Conference on Electronic Computation*, 1960.

[6] B. Davat, Z. Ren, and M. Lajoie-Mazenc. The movement in field modeling. *IEEE Transactions on Magnetics*, 21(6):2296–2298, November 1985.

[7] C.R.I Emson and J. Simkin. An optimal method for 3-d eddy currents. *IEEE Transactions on Magnetics*, 19:2450, 1983.

[8] T.J. Flack and A.F. Volschenk. Computational aspects of time-stepping finite element analysis using an air-gap element. *Proceedings of ICEM'94, Paris*, 1994.

[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] S. Gerber, J.M. Strauss, and P.J. Randewijk. Evaluation of a hybrid finite element analysis package featuring dual air-gap elements. In *Proceedings of the XIX International Conference on Electrical Machines*, Rome, 2010.

[11] H. De Gersem and T. Weiland. A computationally efficient air-gap element for 2-d fe machine models. *IEEE Trans. Magn.*, 41(5):1844–1847, May 2005.

[12] N.E. Gibss, Jr. W.G. Poole, and P.K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM J. Numer. Anal.*, 13(2):236–250, April 1976.

[13] H.A. Haus and J.R. Melcher. *Electromagnetic Fields and Energy.* Prentice-Hall International, Inc., 1989.

[14] Pavel Holoborodko. Numerical integration. http://www.holoborodko.com/pavel/numerical-methods/numerical-integration, October 2008.

[15] A. Hrenikoff. Solution of problems in elasticity by the framework method. *Journal of Applied Mechanics*, 8:169–175, 1941.

[16] K.H. Huebner, D.L. Dewhirst, D.E. Smith, and T.G. Byrom. *The Finite Element Method for Engineers.* John Wiley & Sons Ltd., 4 edition, 2001.

[17] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007.

[18] D. Lin, P. Zhou, and Q.M. Chen. The effects of steel lamination core losses on transient magnetic fields using t-ω; method. In *Vehicle Power and Propulsion Conference, 2008. VPPC '08. IEEE*, pages 1 –4, 3-5 2008.

[19] D. Lin, P. Zhou, W.N. Fu, Z. Badics, and Z.J. Cendes. A dynamic core loss model for soft ferromagnetic and power ferrite materials in transient finite element analysis. *Magnetics, IEEE Transactions on*, 40(2):1318 – 1321, march 2004.

[20] Pearu Peterson. F2py: a tool for connecting fortran and python programs. *Int. J. Comput. Sci. Eng.*, 4(4):296–305, 2009.

[21] D. Rodger, H.C. Lai, and P.J. Leonard. Coupled elements for problems involving movement. *IEEE Trans. Magn.*, 26(2):548–550, March 1990.

[22] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[23] P. Silvester. High-order polynomial triangular elements for potential problems. *International Journal of Engineering Science*, 7:849–861, 1969.

[24] J. Simkin and C.W. Trowbridge. On the use of the total scalar potential in the numerical solution of field problems in electromagnets. *International Journal for Numerical Methods in Engineering*, 14:423, 1979.

[25] J. A. Stegmann and M. J. Kamper. Design aspects of medium power double rotor radial flux air-cored pm wind generators. In *Energy Conversion Congress and Exposition, 2009. ECCE 2009. IEEE*, pages 3634–3640, 2009.

[26] J.A. Stratton. *Electromagnetic Theory*. John Wiley & Sons Inc., 2007.

[27] M.J. Turner, R.W. Clough, H.C. Martin, and L.C. Topp. Stiffness and deflection analysis of complex structures. *Journal of the Aeronautical Sciences*, 23(9):805–823,854, 1956.

[28] G.N. Vanderplaats. *Multidiscipline Design Optimization*. Vanderplaats Research & Development, Inc., 126 Bonifacio Place, Suite F, Monterey, CA 93940, 2007.

[29] R. Wang, H. Mohellebi, T.J. Flack, M.J. Kamper, J.D. Buys, and M. Feliachi. Two-dimensional cartesian air-gap element (cage) for dynamic finite-element modeling of electrical machines with a flat air gap. *IEEE Trans. Magn.*, 38(2):1357–1360, Mar. 2002.

[30] Rong-Jie Wang. *Design aspects and optimisation of an axial field permanent magnet machine with an ironless stator*. PhD thesis, University of Stellenbosch, March 2003.

[31] A.A. Winslow. Numerical solution of the quasi-linear poisson equation in a non-uniform triangular mesh. *Journal of Computational Physics*, 1:149, 1971.

[32] P. Zhou, D. Lin, W.N. Fu, B. Ionescu, and Z.J. Cendes. A general cosimulation approach for coupled field-circuit problems. *Magnetics, IEEE Transactions on*, 42(4):1051 –1054, april 2006.

[33] O. C. Zienkiewicz. *The Finite Element Method*. McGraw-Hill, Maidenhead, England, 3rd edition edition, 1977.

Appendices

Appendix A

The axisymmetric system equation coefficients

The derivation presented here follows that presented by Binns et. al. [3] but with added clarifications.

The governing field equation to be solved is

$$\nabla \times \mathbf{H} = \mathbf{J} \tag{A.1}$$

where $\mathbf{H}$ can be represented in terms of the vector potential $\mathbf{A}$ as follows,

$$\mu \mathbf{H} = \nabla \times \mathbf{A} \tag{A.2}$$

Substituting (A.2) into (A.1) leads to

$$\nabla \times \frac{1}{\mu}(\nabla \times \mathbf{A}) = \mathbf{J} \tag{A.3}$$

from which the residue equation can be constructed,

$$\mathbf{R} = \nabla \times \frac{1}{\mu}(\nabla \times \mathbf{A}) - \mathbf{J} = \mathbf{0} \tag{A.4}$$

From Haus et. al. [13], the curl of the vector $\mathbf{A}$ in the cylindrical coordinate system (r, ϕ, z) is

$$\nabla \times \mathbf{A} = \left(\frac{1}{r}\frac{\partial A_z}{\partial \phi} - \frac{\partial A_\phi}{\partial z}\right)\mathbf{i_r} + \left(\frac{\partial A_r}{\partial z} - \frac{\partial A_z}{\partial r}\right)\mathbf{i}_\phi + \left(\frac{1}{r}\frac{\partial (rA_\phi)}{\partial r} - \frac{1}{r}\frac{\partial A_r}{\partial \phi}\right)\mathbf{i_z} \tag{A.5}$$

In the axisymmetric problem under consideration $A_z = A_r = 0$ and (A.5) simplifies to

$$\nabla \times \mathbf{A} = -\frac{\partial A_\phi}{\partial z}\mathbf{i_r} + \frac{1}{r}\frac{\partial (rA_\phi)}{\partial r}\mathbf{i_z} \tag{A.6}$$

Similarly, $J_z = J_r = 0$ and the vector $\mathbf{J}$ simplifies to

$$\mathbf{J} = J_\phi \mathbf{i}_\phi \tag{A.7}$$

Substituting (A.6) and (A.7) into (A.4), the residual becomes

$$\mathbf{R} = \nabla \times \frac{1}{\mu}\left(-\frac{\partial A_\phi}{\partial z}\mathbf{i_r} + \frac{1}{r}\frac{\partial (rA_\phi)}{\partial r}\mathbf{i_z}\right) - J_\phi \mathbf{i}_\phi \tag{A.8}$$

The weighted residual method requires that

$$\int_\Omega \mathbf{w_i} \cdot \mathbf{R} d\Omega = 0 \tag{A.9}$$

$$\int_\Omega \mathbf{w_i} \cdot \left(\nabla \times \frac{1}{\mu}(\nabla \times \mathbf{A})\right) d\Omega = \int_\Omega \mathbf{w_i} \cdot \mathbf{J} d\Omega \tag{A.10}$$

be satisfied for any $\mathbf{w_i}$. In the equation above, Ω is the problem domain and $\mathbf{w_i}$ is a vector of weighting functions

$$\mathbf{w_i} = \begin{bmatrix} \omega_i \\ \omega_i \\ \omega_i \end{bmatrix} \tag{A.11}$$

In practice only a finite number of weighting functions are used. If the number of weighting functions and the number of unknown parameters defining R are chosen equal, a system of linear equations can be formed,

$$\begin{bmatrix} \int_\Omega \mathbf{w_1} \cdot \mathbf{R} d\Omega \\ \int_\Omega \mathbf{w_2} \cdot \mathbf{R} d\Omega \\ \int_\Omega \mathbf{w_3} \cdot \mathbf{R} d\Omega \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{A.12}$$

and the unknown parameters can be solved.

Binns et al. derive a vector form of Green's theorem [3, p. 454] repeated here for convenience.

$$\int_\Omega \mathbf{F} \cdot \nabla \times \mathbf{G} d\Omega = \int_\Omega \mathbf{G} \cdot \nabla \times \mathbf{F} d\Omega - \int_\Gamma (\mathbf{F} \times \mathbf{G}) \cdot \mathbf{n} d\Gamma \tag{A.13}$$

Referring to (A.10) and (A.13) and identifying $\mathbf{F} = \mathbf{w_i}$ and $\mathbf{G} = \frac{1}{\mu}(\nabla \times \mathbf{A})$, (A.10) can be transformed into

$$\int_\Omega (\nabla \times \mathbf{w_i}) \cdot \frac{1}{\mu}(\nabla \times \mathbf{A}) d\Omega - \int_\Gamma \left(\mathbf{w_i} \times \frac{1}{\mu}(\nabla \times \mathbf{A})\right) \cdot \mathbf{n} d\Gamma = \int_\Omega \mathbf{w_i} \mathbf{J} d\Omega \tag{A.14}$$

This step is necessary in order to ensure that when the problem domain is broken up into elements, the integrand remains finite across element boundaries. (see [3, p.249]) The surface integral term in the above equation can be ignored if the weighting functions is set to zero on the boundary. This is appropriate when Dirichlet ($A_\phi = 0$) boundary conditions are specified. In practice this can be accomplished by deleting the boundary node from the system equation. If Neumann ($\frac{\partial A_\phi}{\partial n} = 0$) boundary conditions are specified, the integral is equal to zero and can be ignored. On boundaries where periodic ($A(z) = A(z + z_0)$) boundary conditions are specified, nodes from the primary boundary are coupled to nodes on the secondary boundary so that the model closes on itself and there is in fact no boundary. When the problem domain is broken up into elements and (A.14) is evaluated element by element, contributions from the surface integral term from neighbouring elements cancel and the only contribution still comes from boundary nodes. Thus, for all the cases described here, the surface integral term is not to be considered and (A.14) becomes

$$\int_\Omega (\nabla \times \mathbf{w_i}) \cdot \frac{1}{\mu}(\nabla \times \mathbf{A})\mathrm{d}\Omega = \int_\Omega \mathbf{w_i}\mathbf{J}\mathrm{d}\Omega \tag{A.15}$$

Expanding the integrands in (A.15) yields

$$\int_\Omega \begin{bmatrix} \frac{1}{r}\frac{\partial \omega_i}{\partial \phi} - \frac{\partial \omega_i}{\partial z} \\ \frac{\partial \omega_i}{\partial z} - \frac{\partial \omega_i}{\partial r} \\ \frac{1}{r}\frac{\partial r\omega_i}{\partial r} - \frac{1}{r}\frac{\partial \omega_i}{\partial \phi} \end{bmatrix} \cdot \frac{1}{\mu} \begin{bmatrix} -\frac{\partial A_\phi}{\partial z} \\ 0 \\ \frac{1}{r}\frac{\partial r A_\phi}{\partial r} \end{bmatrix} \mathrm{d}\Omega = \int_\Omega \begin{bmatrix} \omega_i \\ \omega_i \\ \omega_i \end{bmatrix} \cdot \begin{bmatrix} 0 \\ J_\phi \\ 0 \end{bmatrix} \mathrm{d}\Omega \tag{A.16}$$

In the axisymmetric case under consideration, $\omega_i = \omega_i(r, z)$ and so $\frac{\partial \omega_i}{\partial \phi} = 0$. Thus (A.16) can be simplified to

$$\int_\Omega \frac{1}{\mu}\left(\frac{\partial \omega_i}{\partial z}\frac{\partial A_\phi}{\partial z} + \frac{1}{r^2}\frac{\partial (r\omega_i)}{\partial r}\frac{\partial (rA_\phi)}{\partial r}\right)\mathrm{d}\Omega = \int_\Omega \omega_i J_\phi \mathrm{d}\Omega \tag{A.17}$$

$$\int_\Omega \frac{1}{\mu}\left(\frac{\partial \omega_i}{\partial z}\frac{\partial A_\phi}{\partial z} + \frac{1}{r^2}\frac{\partial (r\omega_i)}{\partial r}\frac{\partial (rA_\phi)}{\partial r}\right) r\mathrm{d}\phi\mathrm{d}r\mathrm{d}z = \int_\Omega \omega_i J_\phi r\mathrm{d}\phi\mathrm{d}r\mathrm{d}z \tag{A.18}$$

Because both integrands in the above equation is independent of ϕ, the integration over $\mathrm{d}\phi$ is equivalent to multiplication be a constant and can be dropped.

Using the finite-element method the problem domain is typically broken up into small triangular elements on which the unknown function A_ϕ is approximated by

$$A_\phi = \sum_{i=1}^{3} N_i u_i^e \tag{A.19}$$

for a first order triangular element with u_i^e the unknown value of the vector potential at node i of the triangle and N_i the shape function

$$N_i = \frac{a_i + b_i r + c_i z}{2A} \qquad \begin{aligned} a_1 &= r_2 z_3 - r_3 z_2 \\ b_1 &= z_2 - z_3 \\ c_1 &= r_3 - r_2 \end{aligned} \tag{A.20}$$

where (r_i, z_i) are the coordinates of node i of the triangular element. The constants a_2, b_2, c_2, a_3, …, etc. are given by cyclic permutations of the suffices shown in (A.20). For a derivation of this shape function and the constants the reader is referred to Binns et. al. [3, p. 247].

By dividing the problem domain Ω into elements, the integrals over the entire problem domain in (A.18) can be replaced by the sum of the integrals over the element domains Ω^e. Considering the integral over a single element and substituting the approximation (A.19) into (A.18) yields

$$\int_{\Omega^e} \frac{1}{\mu} \sum_{j=1}^{3} \left[\left(\frac{\partial \omega_i}{\partial z} \frac{\partial N_j}{\partial z} + \frac{1}{r^2} \frac{\partial (r\omega_i)}{\partial r} \frac{\partial (rN_j)}{\partial r} \right) r u_j^e \right] \mathrm{d}r\mathrm{d}z = \int_{\Omega^e} \omega_i J_\phi r \mathrm{d}r\mathrm{d}z \tag{A.21}$$

which is of the form

$$\mathbf{k_i}^T \mathbf{u} = f_i \tag{A.22}$$

and corresponds to a single row of (A.12), with $\mathbf{u}$ the vector of unknown parameters. For the case of first order triangles with three unknowns, three weighting functions are chosen and the resulting matrix equation is

$$\mathbf{K}^e \mathbf{u}^e = \mathbf{f}^e \tag{A.23}$$

If Galerkin weighting is applied ($\omega_i = N_i$) the coefficients of the matrix $\mathbf{K}^e$ and the vector $\mathbf{f}^e$ are given by

$$K_{ij}^e = \int_{\Omega^e} \frac{1}{\mu} \left(\frac{\partial N_i}{\partial z} \frac{\partial N_j}{\partial z} + \frac{1}{r^2} \frac{\partial (rN_i)}{\partial r} \frac{\partial (rN_j)}{\partial r} \right) r \mathrm{d}r\mathrm{d}z \tag{A.24}$$

$$f_i^e = \int_{\Omega^e} N_i J_\phi r \mathrm{d}r\mathrm{d}z \tag{A.25}$$

Substituting (A.20) into (A.24) and (A.25) and evaluating the partial derivatives yields

$$K_{ij}^e = \int_{\Omega^e} \frac{1}{4A^2\mu} \left[c_i c_j + \frac{1}{r^2}(a_i + 2b_i r + c_i z)(a_j + 2b_j r + c_j z) \right] r \mathrm{d}r\mathrm{d}z \tag{A.26}$$

$$f_i^e = \int_{\Omega^e} J_\phi \left(\frac{a_i + b_i r + c_i z}{2A} \right) r \mathrm{d}r\mathrm{d}z \tag{A.27}$$

which is typically evaluated with a numerical integration scheme such as Gaussian quadrature because performing the integration analytically is cumbersome and neither the accuracy nor the computational cost of the numerical integration prohibits its use.

Appendix B

The axisymmetric Newton-Raphson Jacobian

For the axisymmetric case described in appendix A, the Jacobian matrix used in the Newton-Raphson method needs to be calculated. This derivation does not appear in Binns et. al. [3] but it closely follows the procedure used in the presented derivation for the Cartesian coordinate system.

From Binns et. al. [3], the Jacobian matrix for a single triangular element is given by

$$\mathbf{J}^e = \begin{bmatrix} \frac{\partial F_1}{\partial u_1^e} & \frac{\partial F_1}{\partial u_2^e} & \frac{\partial F_1}{\partial u_3^e} \\ \frac{\partial F_2}{\partial u_1^e} & \frac{\partial F_2}{\partial u_2^e} & \frac{\partial F_2}{\partial u_3^e} \\ \frac{\partial F_1}{\partial u_3^e} & \frac{\partial F_3}{\partial u_2^e} & \frac{\partial F_3}{\partial u_3^e} \end{bmatrix} \tag{B.1}$$

with F_i the residual, namely

$$F_i = K_{i1}^e u_1^e + K_{i2}^e u_2^e + K_{i3}^e u_3^e - Q^e \tag{B.2}$$

with u_i^e the vector potential at node i of the triangular element, Q^e the current density (to avoid confusion) and K_{ij}^e the general term of the stiffness matrix (see appendix A), namely

$$K_{ij}^e = \int_{\Omega^e} \kappa \left(\frac{\partial N_i}{\partial z} \frac{\partial N_j}{\partial z} + \frac{1}{r^2} \frac{\partial (rN_i)}{\partial r} \frac{\partial (rN_j)}{\partial r} \right) d\Omega^e \tag{B.3}$$

with κ the reluctivity $\kappa = \frac{1}{\mu}$. Thus, the general term of the Jacobian matrix is given by

$$J_{ij}^e = \frac{\partial F_i}{\partial u_j^e} = K_{ij}^e + \sum_{h=1}^{3} \frac{\partial K_{ih}^e}{\partial u_j^e} u_h^e \tag{B.4}$$

Consider the term

$$\frac{\partial K_{ih}^e}{\partial u_j^e} u_h^e = \int_{\Omega^e} \frac{\partial \kappa}{\partial u_j^e} \left(\frac{\partial N_i}{\partial z} \frac{\partial N_h}{\partial z} + \frac{1}{r^2} \frac{\partial (rN_i)}{\partial r} \frac{\partial (rN_h)}{\partial r} \right) u_h^e d\Omega^e \tag{B.5}$$

The derivative $\frac{\partial \kappa}{\partial u_j^e}$ is problematic since the reluctivity is not easily related to the vector potential at a single node. However, through the B-H curve the reluctivity is directly related to the magnitude of the flux density. Thus, it is desirable to express the derivative as

$$\frac{\partial \kappa}{\partial u_j^e} = \frac{\partial \kappa}{\partial (p^2)} \frac{\partial (p^2)}{\partial u_j^e} \tag{B.6}$$

where p is the magnitude of the flux density

$$p = ||\mathbf{B}|| = \sqrt{\left(\frac{\partial A_\phi}{\partial z} \right)^2 + \left(\frac{1}{r} \frac{\partial (rA_\phi)}{\partial r} \right)^2} \tag{B.7}$$

Since $\frac{\partial \kappa}{\partial (p^2)}$ is easily obtained in every iteration of the Newton-Raphson method, the only remaining problem is to find $\frac{\partial (p^2)}{\partial u_j^e}$. It is shown here that

$$\begin{aligned}
&\frac{\partial (p^2)}{\partial u_j^e} \\
&= \frac{\partial}{\partial u_j^e} \left[\left(\frac{\partial A_\phi}{\partial z} \right)^2 + \left(\frac{1}{r} \frac{\partial (rA_\phi)}{\partial r} \right)^2 \right] \\
&= \frac{\partial}{\partial u_j^e} \left[\left(\frac{\partial}{\partial z} \sum_{k=1}^{3} N_k u_k^e \right)^2 + \left(\frac{1}{r} \frac{\partial}{\partial r} \sum_{k=1}^{3} rN_k u_k^e \right)^2 \right] \\
&= 2 \left(\frac{\partial}{\partial z} \sum_{k=1}^{3} N_k u_k^e \right) \cdot \frac{\partial}{\partial u_j^e} \left(\frac{\partial}{\partial z} \sum_{k=1}^{3} N_k u_k^e \right) + 2 \left(\frac{1}{r} \frac{\partial}{\partial r} \sum_{k=1}^{3} rN_k u_k^e \right) \frac{\partial}{\partial u_j^e} \left(\frac{1}{r} \frac{\partial}{\partial r} \sum_{k=1}^{3} rN_k u_k^e \right) \\
&= 2 \left(\frac{\partial}{\partial z} \sum_{k=1}^{3} N_k u_k^e \right) \frac{\partial}{\partial z} N_j + \frac{2}{r^2} \left(\frac{\partial}{\partial r} \sum_{k=1}^{3} rN_k u_k^e \right) \frac{\partial}{\partial r} (rN_j) \\
&= 2 \sum_{k=1}^{3} \left(\frac{\partial N_k}{\partial z} \frac{\partial N_j}{\partial z} + \frac{1}{r^2} \frac{\partial (rN_k}{\partial r} \frac{\partial (rN_j)}{\partial r} \right) u_k^e
\end{aligned} \tag{B.8}$$

Substituting (B.6) and (B.5) into (B.4) yields

$$J_{ij}^e = K_{ij}^e + \sum_{h=1}^{3} \int_{\Omega^e} \frac{\partial \kappa}{\partial (p^2)} \frac{\partial (p^2)}{\partial u_h^e} \left(\frac{\partial N_i}{\partial z} \frac{\partial N_h}{\partial z} + \frac{1}{r^2} \frac{\partial (rN_i)}{\partial r} \frac{\partial (rN_h)}{\partial r} \right) u_h^e d\Omega^e \tag{B.9}$$

As in (A.26), we have

$$\frac{\partial N_i}{\partial z} \frac{\partial N_j}{\partial z} + \frac{1}{r^2} \frac{\partial (rN_i)}{\partial r} \frac{\partial (rN_j)}{\partial r} = \frac{1}{4A^2} \underbrace{\left[c_i c_j + \frac{1}{r^2} (a_i + 2b_i r + c_i z)(a_j + 2b_j r + c_j z) \right]}_{t_{ij}} \tag{B.10}$$

with the quantity in brackets being denoted as t_{ij} for the sake of conciseness. Now, using (B.8), (B.9) becomes

$$\begin{aligned} J_{ij}^e &= K_{ij}^e + \sum_{h=1}^{3} \int_{\Omega^e} \frac{\partial \kappa}{\partial (p^2)} \left[2 \sum_{k=1}^{3} \frac{1}{4A^2} t_{kj} u_k^e \right] \cdot \frac{1}{4A^2} t_{ih} u_h^e d\Omega^e \\ &= K_{ij}^e + \frac{1}{8A^4} \cdot \frac{\partial \kappa}{\partial (p^2)} \cdot \sum_{h=1}^{3} \sum_{k=1}^{3} \left(\int_{\Omega^e} t_{kj} t_{ih} r dr dz \right) u_k^e u_h^e \end{aligned} \tag{B.11}$$

where the integral is typically evaluated using Gaussian quadrature.

Appendix C

Complete derivation of the ASAGE

C.1 Air-gap field solution

In an axisymmetric air-gap region, such as illustrated in figure C.1, the magnetic vector potential A_ϕ is governed by

$$\nabla \times (\nabla \times \mathbf{A}) = 0 \tag{C.1}$$

$$\frac{\partial^2 A_\phi}{\partial z^2} + \frac{\partial^2 A_\phi}{\partial r^2} + \frac{1}{r}\frac{\partial A_\phi}{\partial r} - \frac{1}{r^2}A_\phi = 0 \tag{C.2}$$

For the purpose of the air-gap element the following boundary conditions will be imposed on the solution of (C.2),

$$A_\phi(r, z) = A_\phi(r, z + z_0) \tag{C.3}$$

$$A_\phi(a, z) = \sum_i \alpha_i(a, z)u_i^\varepsilon \tag{C.4}$$

$$A_\phi(b, z) = \sum_i \alpha_i(b, z)u_i^\varepsilon \tag{C.5}$$

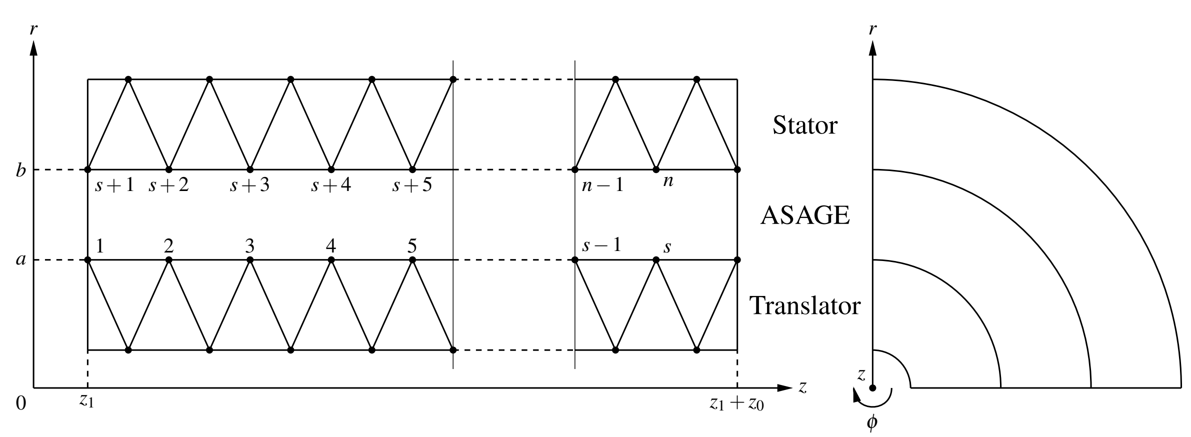


Figure C.1: The domain of the axisymmetric air-gap element

Equation (C.3) represents a positive periodic boundary condition at the sides of the air-gap element while (C.4) and (C.5) state that the solution on the edge of the air-gap element should conform to the solution on the edge of the traditionally meshed regions of the model.

Using separation of variables, let $A_\phi(r,z) = R(r)Z(z)$, then (C.2) becomes

$$\begin{aligned}\frac{\partial^2}{\partial z^2}(R(r)Z(z)) + \frac{\partial^2}{\partial r^2}(R(r)Z(z)) + \frac{1}{r}\frac{\partial}{\partial z}(R(r)Z(z)) - \frac{1}{r^2}R(r)Z(z) = 0\\ RZ'' + R''Z + \frac{1}{r}R'Z - \frac{1}{r^2}RZ = 0\\ RZ'' = Z\left(\frac{1}{r^2}R - \frac{1}{r}R' - R''\right)\\ \frac{Z''}{Z} = \frac{1}{R}\left(\frac{1}{r^2}R - \frac{1}{r}R' - R''\right) = c_\lambda \end{aligned} \tag{C.6}$$

Now there are three cases to evaluate depending on the value of the separation constant c_λ. These cases are

1. $c_\lambda = -\lambda^2$ 2. $c_\lambda = 0$ 3. $c_\lambda = \lambda^2$

The third case will not be considered here since it produces no non-trivial solutions that can satisfy (C.3).

Case 1: $c_\lambda = -\lambda^2$

The equation in Z is

$$Z'' + \lambda^2 Z = 0 \tag{C.7}$$

and its solution is

$$Z(z) = c_1 \cos\lambda z + c_2 \sin\lambda z \tag{C.8}$$

The equation in R is

$$\begin{aligned}R'' + \frac{1}{r}R' - (\lambda^2 + \frac{1}{r^2})R = 0\\ r^2R'' + rR' - (\lambda^2 r^2 + 1)R = 0\end{aligned} \tag{C.9}$$

which is a modified parametric Bessel equation. Its solution is

$$R(r) = c_3 I_1(\lambda r) + c_4 K_1(\lambda r) \tag{C.10}$$

with I_1 and K_1 modified Bessel functions of the 1st and 2nd kind respectively. Thus, from the first case a solution to (C.2) is

$$A_\phi(r,z) = (c_1\cos\lambda z + c_2 \sin\lambda z)(c_3 I_1(\lambda r) + c_4 K_1(\lambda r)) \tag{C.11}$$

Case 2: $c_\lambda = 0$

The equation in Z is

$$Z'' = 0 \tag{C.12}$$

Its solution is

$$Z(z) = c_1 + c_2 z \tag{C.13}$$

The equation in R is

$$\begin{aligned} R'' + \frac{1}{r}R' - \frac{1}{r^2}R &= 0 \\ r^2 R'' + rR' - R &= 0 \end{aligned} \tag{C.14}$$

which is a Cauchy-Euler equation. It can be solved by guessing that the solution takes the form

$$R(r) = r^m \tag{C.15}$$

and then solving for the possible values of m. Substituting (C.15) into (C.14) yields the solution,

$$\begin{aligned} m(m-1)r^m + mr^m - r^m &= 0 \\ r^m(m^2 - m + m - 1) &= 0 \\ m^2 - 1 &= 0 \\ m &= \pm 1 \end{aligned}$$

$$R(r) = c_3 r + c_4 r^{-1} \tag{C.16}$$

Thus, from the second case the solution to (C.2) is

$$\begin{aligned} A_\phi(r,z) &= (c_1 + c_2 z)(c_3 r + c_4 r^{-1}) \\ &= c_1 c_3 r + c_1 c_4 r^{-1} c_2 c_3 r z + c_2 c_4 r^{-1} z \\ &= B_0 r + C_0 r^{-1} + D_0 r z + E_0 r^{-1} z \end{aligned} \tag{C.17}$$

General solution

Combining the solutions of the first two cases yields the general solution to (C.2), namely

$$\begin{aligned} A_\phi(r,z) = B_0 r + C_0 r^{-1} + D_0 r z + E_0 r^{-1} z \\ + \sum_n (c_{1_n} \cos\lambda_n z + c_{2_n} \sin\lambda_n z)(c_{3_n} I_1(\lambda_n r) + c_{4_n} K_1(\lambda_n r)) \end{aligned} \tag{C.18}$$

In order to satisfy (C.3), the constants D_0 and E_0 must be set to zero. The following must also hold,

$$\cos(\lambda_n z) = \cos(\lambda_n (z + z_0)) \qquad \text{and} \qquad \sin(\lambda_n z) = \sin(\lambda_n (z + z_0)) \tag{C.19}$$

This can be accomplished by requiring that

$$\lambda_n = \frac{2\pi n}{z_0} \quad , \quad n \in \mathbf{Z} \tag{C.20}$$

Thus, the general solution to (C.2) satisfying (C.3) is

$$A_\phi(r,z) = B_0 r + C_0 r^{-1} + \sum_{n=1}^{\infty}(D_n \cos\lambda_n z + E_n \sin\lambda_n z)(F_n I_1(\lambda_n r) + G_n K_1(\lambda_n r)) \tag{C.21}$$

In order to simplify the satisfaction of (C.4) and (C.5), $A_\phi(r,z)$ is expressed as

$$A_\phi(r,z) = A_{\phi_1}(r,z) + A_{\phi_2}(r,z) \tag{C.22}$$

and the boundary conditions on $A_{\phi_1}(r,z)$ and $A_{\phi_2}(r,z)$ are defined as

$$A_{\phi_1}(a,z) = A_\phi(a,z) \tag{C.23}$$

$$A_{\phi_1}(b,z) = 0 \tag{C.24}$$

and

$$A_{\phi_2}(a,z) = 0 \tag{C.25}$$

$$A_{\phi_2}(b,z) = A_\phi(b,z) \tag{C.26}$$

In this way, the sum of $A_{\phi_1}(r,z)$ and $A_{\phi_2}(r,z)$ have boundary values equal to that of $A_\phi(r,z)$, but the boundary conditions imposed on $A_{\phi_1}(r,z)$ and $A_{\phi_2}(r,z)$ are satisfied more easily.

The function $A_{\phi_1}(r,z)$ is defined similar to (C.21), i.e.

$$A_{\phi_1}(r,z) = B_{10} r + C_{10} r^{-1} + \sum_{n=1}^{\infty}(D_{1n} \cos\lambda_n z + E_{1n} \sin\lambda_n z)(F_{1n} I_1(\lambda_n r) + G_{1n} K_1(\lambda_n r)) \tag{C.27}$$

Setting $A_{\phi_1}(b,z)$ equal to zero in order to satisfy (C.24) yields

$$A_{\phi_1}(b,z) = 0 = B_{10} b + C_{10} b^{-1} + \sum_{n=1}^{\infty}(D_{1n} \cos\lambda_n z + E_{1n} \sin\lambda_n z)(F_{1n} I_1(\lambda_n b) + G_{1n} K_1(\lambda_n b)) \tag{C.28}$$

$$B_{10} b + C_{10} b^{-1} = 0 \qquad\qquad F_{1n} I_1(\lambda_n b) + G_{1n} K_1(\lambda_n b) = 0$$

$$C_{10} = -B_{10} b^2 \qquad\qquad G_{1n} = -F_{1n}\frac{I_1(\lambda_n b)}{K_1(\lambda_n b)}$$

Substituting the values of C_{10} and G_{1n} back into (C.28) yields

$$\begin{aligned} A_{\phi_1}(r,z) = {} & B_{10} r - B_{10}\frac{b^2}{r} \\ & + \sum_{n=1}^{\infty}(D_{1n} \cos\lambda_n z + E_{1n} \sin\lambda_n z)\left(F_{1n} I_1(\lambda_n r) - F_{1n}\frac{I_1(\lambda_n b)}{K_1(\lambda_n b)} K_1(\lambda_n r)\right) \end{aligned} \tag{C.29}$$

and by making the following substitutions

$$a_{10} = B_{10} \qquad a_{1n} = D_{1n}F_{1n} \qquad b_{1n} = E_{1n}F_{1n} \tag{C.30}$$

equation C.29 can be further simplified to

$$A_{\phi_1}(r,z) = a_{10}\left(r - \frac{b^2}{r}\right) + \sum_{n=1}^{\infty}\left(I_1(\lambda_n r) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)}K_1(\lambda_n r)\right)(a_{1n}\cos\lambda_n z + b_{1n}\sin\lambda_n z) \tag{C.31}$$

Following the same procedure to satisfy (C.25), $A_{\phi_2}(r,z)$ is given by

$$A_{\phi_2}(r,z) = a_{20}\left(r - \frac{a^2}{r}\right) + \sum_{n=1}^{\infty}\left(I_1(\lambda_n r) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)}K_1(\lambda_n r)\right)(a_{2n}\cos\lambda_n z + b_{2n}\sin\lambda_n z) \tag{C.32}$$

Since $A_\phi(a,z) = A_{\phi_1}(a,z)$ and $A_\phi(b,z) = A_{\phi_2}(b,z)$, the solution on the boundaries at $r = a$ and $r = b$ is

$$A_\phi(a,z) = a_{10}\left(a - \frac{b^2}{a}\right) + \sum_{n=1}^{\infty}\left(I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)}K_1(\lambda_n a)\right)(a_{1n}\cos\lambda_n z + b_{1n}\sin\lambda_n z) \tag{C.33}$$

$$A_\phi(b,z) = a_{20}\left(b - \frac{a^2}{b}\right) + \sum_{n=1}^{\infty}\left(I_1(\lambda_n b) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)}K_1(\lambda_n b)\right)(a_{2n}\cos\lambda_n z + b_{2n}\sin\lambda_n z) \tag{C.34}$$

The next step is to express the solution on the boundaries at $r = a$ and $r = b$ in terms of the vector potential at the nodes on these boundaries. This expression should take a similar form to (C.33) and (C.34) in order to allow coefficients to be equated. Wang [30] showed in his derivation of the CAGE that the solution on the boundaries of the CAGE at $y = a$ and $y = b$ could be expressed in terms of a Fourier series and the nodal values on the boundary. Although this derivation was done in the Cartesian coordinate system, it is also valid in the cylindrical coordinate system used with the ASAGE with x equivalent to z and y equivalent to r. The resulting forms of equation (C.4) and (C.5) are

$$A_\phi(a,z) = \sum_{i=1}^{s} u_i^\varepsilon\left[\frac{1}{2}a_{0i} + \sum_{n=1}^{\infty}(a_{ni}\cos\lambda_n z + b_{ni}\sin\lambda_n z)\right] \tag{C.35}$$

and

$$A_\phi(b,z) = \sum_{j=s+1}^{t} u_j^\varepsilon\left[\frac{1}{2}a_{0j} + \sum_{n=1}^{\infty}(a_{nj}\cos\lambda_n z + b_{nj}\sin\lambda_n z)\right] \tag{C.36}$$

with the coefficients a_{0i}, a_{ni} and b_{ni} given by

$$a_{0i} = \frac{z_{i+1} - z_{i-1}}{z_0} \tag{C.37}$$

$$\begin{aligned} a_{ni} = -\frac{4}{z_0} \cdot \frac{1}{\lambda_n^2} \Bigg[& \frac{1}{z_i - z_{i-1}} \sin\left(\frac{\lambda_n}{2}(z_i + z_{i-1})\right) \sin\left(\frac{\lambda_n}{2}(z_i - z_{i-1})\right) + \\ & \frac{1}{z_i - z_{i+1}} \sin\left(\frac{\lambda_n}{2}(z_i + z_{i+1})\right) \sin\left(\frac{\lambda_n}{2}(z_i - z_{i+1})\right) \Bigg] \end{aligned} \tag{C.38}$$

$$\begin{aligned} b_{ni} = -\frac{4}{z_0} \cdot \frac{1}{\lambda_n^2} \Bigg[& \frac{1}{z_i - z_{i-1}} \sin\left(\frac{\lambda_n}{2}(z_i - z_{i-1})\right) \cos\left(\frac{\lambda_n}{2}(z_i + z_{i-1})\right) + \\ & \frac{1}{z_i - z_{i+1}} \sin\left(\frac{\lambda_n}{2}(z_{i+1} - z_i)\right) \cos\left(\frac{\lambda_n}{2}(z_{i+1} + z_i)\right) \Bigg] \end{aligned} \tag{C.39}$$

Through comparison of the coefficients of (C.33) and (C.35), it can be seen that

$$a_{10}\left(a - \frac{b^2}{a}\right) = \sum_{i=1}^{s} u_i^{\varepsilon} \frac{1}{2} a_{0i}$$

$$a_{10} = \frac{1}{2(a - \frac{b^2}{a})} \sum_{i=1}^{s} a_{0i} u_i^{\varepsilon} \tag{C.40}$$

and

$$a_{1n}\left(I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1 \lambda_n b} K_1(\lambda_n a)\right) = \sum_{i=1}^{s} a_{ni} u_i^{\varepsilon}$$

$$a_{1n} = \frac{\sum_{i=1}^{s} a_{ni} u_i^{\varepsilon}}{I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)} K_1(\lambda_n a)} \tag{C.41}$$

By similar comparisons

$$a_{2n} = \frac{\sum_{j=s+1}^{t} a_{nj} u_j^{\varepsilon}}{I_1(\lambda_n b) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)} K_1(\lambda_n b)} \tag{C.42}$$

$$b_{1n} = \frac{\sum_{i=1}^{s} b_{ni} u_i^{\varepsilon}}{I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)} K_1(\lambda_n a)} \tag{C.43}$$

$$b_{2n} = \frac{\sum_{j=s+1}^{t} b_{nj} u_j^{\varepsilon}}{I_1(\lambda_n b) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)} K_1(\lambda_n b)} \tag{C.44}$$

Substituting (C.40) through (C.44) back into (C.31) and (C.32), (C.22) becomes

$$\begin{aligned} A_\phi(r, z) = \sum_{i=1}^{s} u_i^{\varepsilon} \Bigg[& \frac{r - \frac{b^2}{r}}{2(a - \frac{b^2}{a})} a_{0i} \\ & + \sum_{n=1}^{\infty} \left(\frac{I_1(\lambda_n r) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)} K_1(\lambda_n r)}{I_1(\lambda_n a) - \frac{I_1(\lambda_n b)}{K_1(\lambda_n b)} K_1(\lambda_n a)} \right) (a_{ni} \cos \lambda_n z + b_{ni} \sin \lambda_n z) \Bigg] \\ + \sum_{j=s+1}^{t} u_j^{\varepsilon} \Bigg[& \frac{r - \frac{a^2}{r}}{2(b - \frac{a^2}{b})} a_{0j} \\ & + \sum_{n=1}^{\infty} \left(\frac{I_1(\lambda_n r) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)} K_1(\lambda_n r)}{I_1(\lambda_n b) - \frac{I_1(\lambda_n a)}{K_1(\lambda_n a)} K_1(\lambda_n b)} \right) (a_{nj} \cos \lambda_n z + b_{nj} \sin \lambda_n z) \Bigg] \end{aligned} \tag{C.45}$$

This can be written in a more compact form (as is clearly necessary ...)

$$A_\phi(r,z) = \sum_{i=1}^{t} \alpha_i^\varepsilon(r,z) u_i^\varepsilon \tag{C.46}$$

with

$$\alpha_i^\varepsilon(r,z) = \frac{r - \frac{c_i^2}{r}}{c_i' - \frac{c_i^2}{c_i'}} \frac{a_{0i}}{2} + \sum_{n=1}^{\infty} \left(\frac{I_1(\lambda_n r) - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n r)}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n c_i')} \right) (a_{ni} \cos \lambda_n z + b_{ni} \sin \lambda_n z) \tag{C.47}$$

$$\text{where} \begin{cases} c_i = b \quad \text{and} \quad c_i' = a \quad \text{if} \quad i \epsilon \{1, 2, \ldots, s\} \\ c_i = a \quad \text{and} \quad c_i' = b \quad \text{if} \quad i \epsilon \{s+1, \ldots, t\} \end{cases} \tag{C.48}$$

C.2 Stiffness matrix

In the axisymmetric case under consideration, the weighted residual method gives the following formula for the minimisation of the residual,

$$\int_{\Omega^\varepsilon} \left(\frac{\partial \omega_i}{\partial z} \frac{\partial A_\phi}{\partial z} + \frac{1}{r^2} \frac{\partial (r\omega_i)}{\partial r} \frac{\partial (rA_\phi)}{\partial r} \right) \mathrm{d}\Omega^\varepsilon = 0 \tag{C.49}$$

with ω_i an arbitrary weighting function and Ω^ε the domain of the air-gap element. Using Galerkin weighting, the weighting functions are chosen as the shape functions ($\omega_i = \alpha_i^\varepsilon$) and by substituting (C.47) into (C.49) the expression for the i'th row of the system equation becomes

$$R_i = \int_{\Omega^\varepsilon} \sum_{j=1}^{t} \left[\left(\frac{\partial \alpha_i^\varepsilon}{\partial z} \frac{\partial \alpha_j^\varepsilon}{\partial z} + \frac{1}{r^2} \frac{\partial (r\alpha_i^\varepsilon)}{\partial r} \frac{\partial (r\alpha_j^\varepsilon)}{\partial r} \right) u_j^\varepsilon \right] \mathrm{d}\Omega^\varepsilon = 0 \tag{C.50}$$

resulting in the general term of the air-gap element's stiffness matrix being given by

$$K_{ij}^\varepsilon = \int_{\Omega^\varepsilon} \left(\frac{\partial \alpha_i^\varepsilon}{\partial z} \frac{\partial \alpha_j^\varepsilon}{\partial z} + \frac{1}{r^2} \frac{\partial (r\alpha_i^\varepsilon)}{\partial r} \frac{\partial r\alpha_j^\varepsilon}{\partial r} \right) r \mathrm{d}r \mathrm{d}z \tag{C.51}$$

For the sake of conciseness, it is useful to express $\alpha_i^\varepsilon(r,z)$ as

$$\begin{aligned} \alpha_i^\varepsilon(r,z) &= \underbrace{\frac{r - \frac{c_i^2}{r}}{c_i' - \frac{c_i^2}{c_i'}} \frac{a_{0i}}{2}}_{f_{1i}(r)} + \sum_{n=1}^{\infty} \underbrace{\left(\frac{I_1(\lambda_n r) - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n r)}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n c_i')} \right)}_{f_{2i}(r)} \underbrace{(a_{ni} \cos \lambda_n z + b_{ni} \sin \lambda_n z)}_{g_i(z)} \\ &= f_{1i}(r) + \sum_{n=1}^{\infty} f_{2i}(r) g_i(z) \end{aligned} \tag{C.52}$$

Here follows the simplification of (C.51). The following identities taken from Wang [30] are useful,

$$\int_{z_1}^{z_1+z_0} \sin \lambda_n z \mathrm{d}z = 0 \tag{C.53}$$

$$\int_{z_1}^{z_1+z_0} \cos \lambda_n z \mathrm{d}z = 0 \tag{C.54}$$

$$\int_{z_1}^{z_1+z_0} \sin \lambda_n z \cdot \cos \lambda_m z \mathrm{d}z = 0 \tag{C.55}$$

$$\int_{z_1}^{z_1+z_0} \sin \lambda_n z \cdot \sin \lambda_m z \mathrm{d}z = \xi \tag{C.56}$$

$$\int_{z_1}^{z_1+z_0} \cos \lambda_n z \cdot \cos \lambda_m z \mathrm{d}z = \xi \tag{C.57}$$

with $\lambda_n = \frac{2\pi n}{z_0}$, $\lambda_m = \frac{2\pi m}{z_0}$ and ξ such that

$$\xi = \begin{cases} 0 & \text{when} \quad n \neq m \\ \frac{z_0}{2} & \text{when} \quad n = m \end{cases} \tag{C.58}$$

The first term in (C.51) is

$$\begin{aligned} &\int_{\Omega^\varepsilon} \frac{\partial \alpha_i^\varepsilon}{\partial z} \frac{\partial \alpha_j^\varepsilon}{\partial z} \mathrm{d}\Omega^\varepsilon \\ &= \int_{\Omega^\varepsilon} \sum_{n=1}^{\infty} f_{2i}(r) g_i'(z) \cdot \sum_{m=1}^{\infty} f_{2j}(r) g_j'(z) \mathrm{d}\Omega^\varepsilon \\ &= \int_{r=a}^{b} \int_{z=z_1}^{z_1+z_0} \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} f_{2i}(r) f_{2j}(r) \\ &\quad \cdot (-a_{ni}\lambda_n \sin \lambda_n z + b_{ni}\lambda_n \cos \lambda_n z)(-a_{mj}\lambda_m \sin \lambda_m z + b_{mj}\lambda_m \cos \lambda_m z) r \mathrm{d}z \mathrm{d}r \end{aligned} \tag{C.59}$$

Using the identities (C.55) through (C.57), (C.59) becomes

$$\begin{aligned} &\int_{\Omega^\varepsilon} \frac{\partial \alpha_i^\varepsilon}{\partial z} \frac{\partial \alpha_j^\varepsilon}{\partial z} \mathrm{d}\Omega^\varepsilon \\ &= \int_{r=a}^{b} \sum_{n=1}^{\infty} f_{2i}(r) f_{2j}(r) \left(a_{ni}a_{nj}\lambda_n^2 \frac{z_0}{2} + b_{ni}b_{nj}\lambda_n^2 \frac{z_0}{2} \right) r \mathrm{d}r \\ &= \frac{z_0}{2} \sum_{n=1}^{\infty} \left(\lambda_n^2 (a_{ni}a_{nj} + b_{ni}b_{nj}) \cdot \int_a^b f_{2i}(r) f_{2j}(r) r \mathrm{d}r \right) \end{aligned} \tag{C.60}$$

The second term of (C.51) requires a bit more work. Consider first the derivative

$$\begin{aligned} \frac{\partial (r\alpha_i^\varepsilon)}{\partial r} &= \frac{\partial}{\partial r} \left(r f_{1i}(r) + \sum_{n=1}^{\infty} r f_{2i}(r) g_i(z) \right) \\ &= f_{1i} + r f_{1i}'(r) + \sum_{n=1}^{\infty} \left[f_{2i}(r) + r f_{2i}'(r) \right] g_i(z) \end{aligned} \tag{C.61}$$

$$\begin{aligned} f_{1i}(r) + r f_{1i}'(r) &= \frac{r - \frac{c_i^2}{r}}{c_i' - \frac{c_i^2}{c_i'}} + r \left(\frac{1 + \frac{c_i^2}{r^2}}{c_i' - \frac{c_i^2}{c_i'}} \frac{a_0 i}{2} \right) \\ &= \frac{r - \frac{c_i^2}{r} + r + \frac{c_i^2}{r}}{c_i' - \frac{c_i^2}{c_i'}} \cdot \frac{a_{0i}}{2} \\ &= \underbrace{\frac{r}{c_i' - \frac{c_i^2}{c_i'}} \cdot a_{0i}}_{f_{3i}(r)} \end{aligned} \tag{C.62}$$

The modified Bessel functions used here have the following properties,

$$\frac{\partial}{\partial r} I_1(\lambda_n r) = \lambda_n I_0(\lambda_n r) - \frac{1}{r} I_1(\lambda_n r) \tag{C.63}$$

$$\frac{\partial}{\partial r} K_1(\lambda_n r) = -\lambda_n K_0(\lambda_n r) - \frac{1}{r} K_1(\lambda_n r) \tag{C.64}$$

Using (C.63) and (C.64),

$$\begin{aligned} & f_{2i}(r) + r f_{2i}'(r) \\ &= \frac{I_1(\lambda_n r) - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n r)}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n c_i')} \\ &\quad + \frac{r \left[\lambda_n I_0(\lambda_n r) - \frac{1}{r} I_1(\lambda_n r) + \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} \left[\lambda_n K_0(\lambda_n r) + \frac{1}{r} K_1(\lambda_n r) \right] \right]}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n c_i')} \\ &= \underbrace{r \lambda_n \left(\frac{I_0(\lambda_n r) + \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_0(\lambda_n r)}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n c_i')} \right)}_{f_{4i}(r)} \end{aligned} \tag{C.65}$$

Thus, substituting (C.62) and (C.65) back into (C.61) one obtains

$$\begin{aligned} \frac{\partial (r \alpha_i^{\varepsilon})}{\partial r} &= \frac{r}{c_i' - \frac{c_i^2}{c_i'}} \cdot a_{0i} + \sum_{n=1}^{\infty} r \lambda_n \left(\frac{I_0(\lambda_n r) + \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_0(\lambda_n r)}{I_1(\lambda_n c_i') - \frac{I_1(\lambda_n c_i)}{K_1(\lambda_n c_i)} K_1(\lambda_n c_i')} \right) (a_{ni} \cos \lambda_n z + b_{ni} \sin \lambda_n z) \\ &= f_{3i}(r) + \sum_{n=1}^{\infty} f_{4i}(r) g_i(z) \end{aligned} \tag{C.66}$$

Now, the second term of (C.51) can be considered

$$\begin{aligned}
&\int_{\Omega^\varepsilon} \frac{1}{r^2}\frac{\partial(r\alpha_i^\varepsilon)}{\partial r}\frac{\partial(r\alpha_j^\varepsilon)}{\partial r}\mathrm{d}\Omega^\varepsilon \\
&= \int_{\Omega^\varepsilon} \frac{1}{r^2}\left[f_{3i}(r) + \sum_{n=1}^{\infty} f_{4i}(r)g_i(z)\right]\left[f_{3j}(r) + \sum_{n=1}^{\infty} f_{4j}(r)g_j(z)\right]\mathrm{d}\Omega^\varepsilon \\
&= \int_{r=a}^{b}\int_{z=z_1}^{z_1+z_0} \frac{1}{r^2} f_{3i}(r)f_{3j}(r)r\mathrm{d}r\mathrm{d}z \\
&\quad + \left.\int_{r=a}^{b}\int_{z=z_1}^{z_1+z_0} \frac{1}{r^2} f_{3i}(r)\sum_{n=1}^{\infty} f_{4j}(r)g_j(z)r\mathrm{d}r\mathrm{d}z\right\} = 0 \\
&\quad + \left.\int_{r=a}^{b}\int_{z=z_1}^{z_1+z_0} \frac{1}{r^2} f_{3j}(r)\sum_{n=1}^{\infty} f_{4i}(r)g_i(z)r\mathrm{d}r\mathrm{d}z\right\} = 0 \\
&\quad + \left.\int_{r=a}^{b}\int_{z=z_1}^{z_1+z_0} \frac{1}{r^2} \sum_{n=1}^{\infty}\sum_{m=1}^{\infty} f_{4i}(r)f_{4j}(r)g_i(z)g_j(z)r\mathrm{d}r\mathrm{d}z\right\}\ T_4
\end{aligned} \tag{C.67}$$

In the above equation, the two middle terms are zero because of (C.53) and (C.54). The integration over z of the first term amounts to multiplication by z_0. Considering the fourth term, we have

$$\begin{aligned}
T_4 = &\int_{r=a}^{b}\int_{z=z_1}^{z_1+z_0} \frac{1}{r^2}\sum_{n=1}^{\infty}\sum_{m=1}^{\infty} f_{4i}(r)f_{4j}(r)(a_{ni}a_{mj}\cos\lambda_n z\cos\lambda_m z \\
&+ a_{ni}b_{mj}\cos\lambda_n z\sin\lambda_m z + b_{ni}a_{mj}\sin\lambda_n z\cos\lambda_m z + b_{ni}b_{mj}\sin\lambda_n z\sin\lambda_m z)r\mathrm{d}r\mathrm{d}z
\end{aligned} \tag{C.68}$$

Once again, using (C.55) through (C.57), this becomes

$$\begin{aligned}
T_4 &= \int_a^b \frac{1}{r^2}\sum_{n=1}^{\infty} f_{4i}(r)f_{4j}(r)(a_{ni}a_{nj}\frac{z_0}{2} + b_{ni}b_{nj}\frac{z_0}{2})r\mathrm{d}r \\
&= \frac{z_0}{2}\sum_{n=1}^{\infty}\left[(a_{ni}a_{nj} + b_{ni}b_{nj})\int_a^b \frac{1}{r^2} f_{4i}(r)f_{4j}(r)r\mathrm{d}r\right]
\end{aligned} \tag{C.69}$$

Substituting (C.69) back into (C.67) yields

$$\begin{aligned}
&\int_{\Omega^\varepsilon} \frac{1}{r^2}\frac{\partial(r\alpha_i^\varepsilon)}{\partial r}\frac{\partial(r\alpha_j^\varepsilon)}{\partial r}\mathrm{d}\Omega^\varepsilon \\
&= z_0\int_a^b \frac{1}{r^2} f_{3i}(r)f_{3j}(r)r\mathrm{d}r + \frac{z_0}{2}\sum_{n=1}^{\infty}\left[(a_{ni}a_{nj} + b_{ni}b_{nj})\int_a^b \frac{1}{r^2} f_{4i}(r)f_{4j}(r)r\mathrm{d}r\right] \\
&= z_0\int_a^b \frac{1}{r^2}\cdot\frac{r}{c_i' - \frac{c_i^2}{c_i'}}\cdot\frac{r}{c_j' - \frac{c_j^2}{c_j'}}\cdot a_{0i}a_{0j}r\mathrm{d}r + \frac{z_0}{2}\sum_{n=1}^{\infty}\left[(a_{ni}a_{nj} + b_{ni}b_{nj})\int_a^b \frac{1}{r^2} f_{4i}(r)f_{4j}(r)r\mathrm{d}r\right] \\
&= z_0\cdot\frac{1}{c_i' - \frac{c_i^2}{c_i'}}\cdot\frac{1}{c_j' - \frac{c_j^2}{c_j'}}\cdot a_{0i}a_{0j}\cdot\left.\frac{r^2}{2}\right|_{r=a}^{b} + \frac{z_0}{2}\sum_{n=1}^{\infty}\left[(a_{ni}a_{nj} + b_{ni}b_{nj})\int_a^b \frac{1}{r^2} f_{4i}(r)f_{4j}(r)r\mathrm{d}r\right] \\
&= \frac{z_0(b^2 - a^2)}{2(c_i' - \frac{c_i^2}{c_i'})(c_j' - \frac{c_j^2}{c_j'})}\cdot a_{0i}a_{0j} + \frac{z_0}{2}\sum_{n=1}^{\infty}\left[(a_{ni}a_{nj} + b_{ni}b_{nj})\int_a^b \frac{1}{r^2} f_{4i}(r)f_{4j}(r)r\mathrm{d}r\right]
\end{aligned} \tag{C.70}$$

Finally, substituting (C.60) and (C.70) back into (C.51) gives the expression for the general term of the stiffness matrix

$$K^{\varepsilon}_{ij} = \frac{z_0(b^2 - a^2)}{2(c'_i - \frac{c_i^2}{c'_i})(c'_j - \frac{c_j^2}{c'_j})} \cdot a_{0i}a_{0j} + \frac{z_0}{2}\sum_{n=1}^{\infty}(a_{ni}a_{nj} + b_{ni}b_{nj})\int_a^b \left[\lambda_n^2 f_{2i}(r)f_{2j}(r)r\mathrm{d}r + \frac{1}{r^2}f_{4i}(r)f_{4j}(r)\right] r\mathrm{d}r \quad \text{(C.71)}$$

where the integral is evaluated numerically and the infinite series is truncated at some point.

Appendix D

Force calculation using the ASAGE

In section 5.5, the derivation of a formula used to calculate the force on the translator, in the direction of movement, in tubular axisymmetric problems is given. This formula, repeated here, gives the force in terms of the vector potentials at the boundary of the axisymmetric air-gap element as

$$F_z = -\frac{z_0\pi}{r\mu}\sum_{i=1}^{t}\sum_{j=1}^{t} u_i^{\varepsilon} u_j^{\varepsilon} \sum_{n=1}^{\infty} \lambda_n f_{2i}(r) f_{4j}(r)(b_{ni}a_{nj} - a_{ni}b_{nj}) \tag{D.1}$$

The values of the coefficients a_{ni} and b_{ni} as well as the expressions for the functions $f_{2i}(r)$ and $f_{4j}(r)$ can be obtained in appendix C.

The code implementing this calculation is given in the following listing.

```
subroutine asage_force(m, mage, torq)
  use age_module
  use mesh_module
  use core_input_module
  implicit none

  type(mesh) m
  type(age) mage
  integer i, j, n
  double precision lan, tij, torq, r
  double precision, dimension(mage%nfour,2) :: term_r, term_z

  r = (mage%bot_gap + mage%top_gap)/2d0
  torq=0.0d0

  do n = 1, mage%nfour
    lan = mage%la(n)
```

```
      term_r(n,1) = lan * (mage%I1r(n)*mage%K1b(n) &
        - mage%I1b(n)*mage%K1r(n)) / (mage%I1a(n)*mage%K1b(n) &
        - mage%I1b(n)*mage%K1a(n))

      term_z(n,1) = r*lan * (mage%I0r(n)*mage%K1b(n) &
        + mage%I1b(n)*mage%K0r(n)) / (mage%I1a(n)*mage%K1b(n) &
        - mage%I1b(n)*mage%K1a(n))

      term_r(n,2) = lan * (mage%I1r(n)*mage%K1a(n) &
        - mage%I1a(n)*mage%K1r(n)) / (mage%I1b(n)*mage%K1a(n) &
        - mage%I1a(n)*mage%K1b(n))

      term_z(n,2) = r*lan * (mage%I0r(n)*mage%K1a(n) &
        + mage%I1a(n)*mage%K0r(n)) / (mage%I1b(n)*mage%K1a(n) &
        - mage%I1a(n)*mage%K1b(n))
    end do

    do i = 1, mage%nraz
      do j = 1, mage%nraz
        tij=0.0d0
        do n = 1, mage%nfour
          if (i <= mage%nrs .and. j <= mage%nrs) then
            tij = tij + term_r(n,1)*term_z(n,1) * &
            (mage%bn(n,i)*mage%an(n,j) - mage%an(n,i)*mage%bn(n,j))
          else if (i <= mage%nrs .and. j > mage%nrs) then
            tij = tij + term_r(n,1)*term_z(n,2) * &
            (mage%bn(n,i)*mage%an(n,j) - mage%an(n,i)*mage%bn(n,j))
          else if (i > mage%nrs .and. j <= mage%nrs) then
            tij = tij + term_r(n,2)*term_z(n,1) * &
            (mage%bn(n,i)*mage%an(n,j) - mage%an(n,i)*mage%bn(n,j))
          else if (i > mage%nrs .and. j > mage%nrs) then
            tij = tij + term_r(n,2)*term_z(n,2) * &
            (mage%bn(n,i)*mage%an(n,j) - mage%an(n,i)*mage%bn(n,j))
          end if
        end do
        torq = torq + m%a(mage%razind(i))*m%a(mage%razind(j))*tij
      end do
    end do
    torq = -torq*np*pi*mage%thao/u0
  end subroutine
```

Listing D.1: Implementation of the force calculation for an axisymmetric air-gap element.

Appendix E

Gaussian quadrature

E.1 Introduction

Many of the calculations described in this work require the numerical evaluation of integrals. An efficient way of evaluating these integrals is necessary in order to minimise the cost of the numerical integration. Binns et. al. [3] and Zienkiewicz [33] recommend using Gaussian quadrature. The discussion presented here serves only to illustrate how this method was used in this work. The interested reader is referred to the book by Zienkiewicz where a more thorough discussion of the theory is presented.

Gaussian quadrature is a numerical integration scheme whereby an integral is evaluated by a weighted sum of values of the integrand as follows

$$I = \int_{-1}^{1} f(\xi)\mathrm{d}\xi = \sum_{i=1}^{n} H_i f(\xi_i) \tag{E.1}$$

What makes Gaussian quadrature special is that both the points, ξ_i, and the weights, H_i, are variable. This is in contrast to other integration methods such as the trapezoidal rule or Simpson's rule where the points are fixed and the weights are determined. This means that for the same number of sampling points, Gaussian quadrature can approximate higher degree polynomials, seeing as there are more unknowns. Thus, Gaussian quadrature is an efficient method of evaluating integrals numerically. Determining the values of the points and weights is not simple, but these are reported in the literature (for example [3] and [33]) for numerous cases. The work of Holoborodko [14] was also consulted in this regard.

E.2 One-dimensional quadrature

The points and weights reported in the literature are usually for the domain $\xi \in [-1, 1]$ for one-dimensional quadrature. In this case an integral such as

$$I_1 = \int_a^b f(x)\mathrm{d}x \tag{E.2}$$

must be mapped to (E.1). This is accomplished by the substitutions

$$x = \frac{b-a}{2}\xi + \frac{a+b}{2} \tag{E.3}$$

and

$$\mathrm{d}x = \frac{b-a}{2}\mathrm{d}\xi \tag{E.4}$$

This yields the required I_1 in terms of the known points and weights and the boundaries of integration, namely

$$\begin{aligned} I_1 &= \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}\xi + \frac{a+b}{2}\right)\mathrm{d}\xi \\ &= \frac{b-a}{2}\sum_{i=1}^{n} H_i f\left(\frac{b-a}{2}\xi_i + \frac{a+b}{2}\right) \end{aligned} \tag{E.5}$$

This formula was used in the evaluation of (5.44).

E.3 Quadrature over triangles

For triangles, Gaussian quadrature points and weights are usually available in terms of triangular area coordinates. The formula used to evaluate the integrals in (5.12), (5.13) and (5.49) is

$$I_T = \int_A f(\xi)\mathrm{d}a = A\sum_{i=1}^{n} H_i f(\xi_i^1, \xi_i^2, \xi_i^3) \tag{E.6}$$

where A is the area of the triangle and ξ_i^1, ξ_i^2 and ξ_i^3 are the area coordinates.

Appendix F

Screenshots from field plot viewer

A couple of screen shots from the Python application developed to view field plot files is shown. The application is capable of displaying element numbers, node numbers, variable numbers, elemental flux densities (figure F.3), nodal vector potentials, flux lines (figures F.1 and F.2), the mesh, colour coded material types (figure F.2) and a colour map of the flux density (figures F.1 and F.3). The application is also capable of locating a specific element or node by number – a useful feature for debugging purposes.

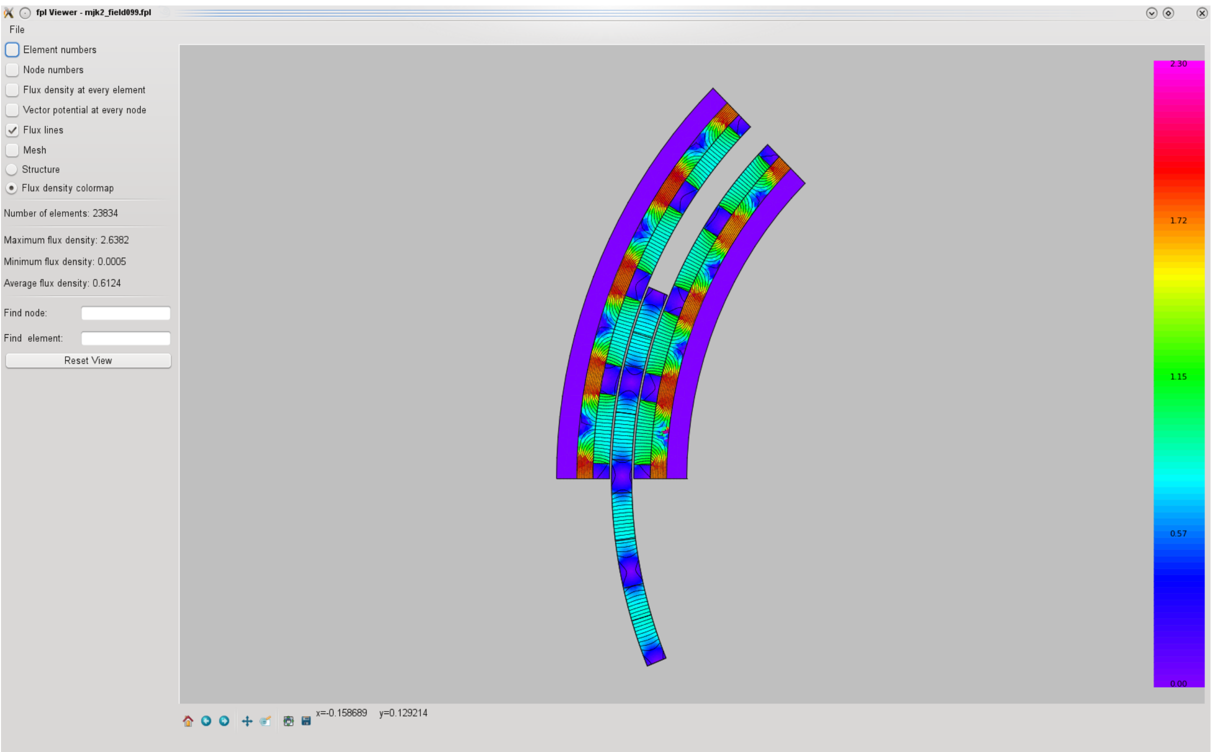


Figure F.1: Flux density colour map with movement.

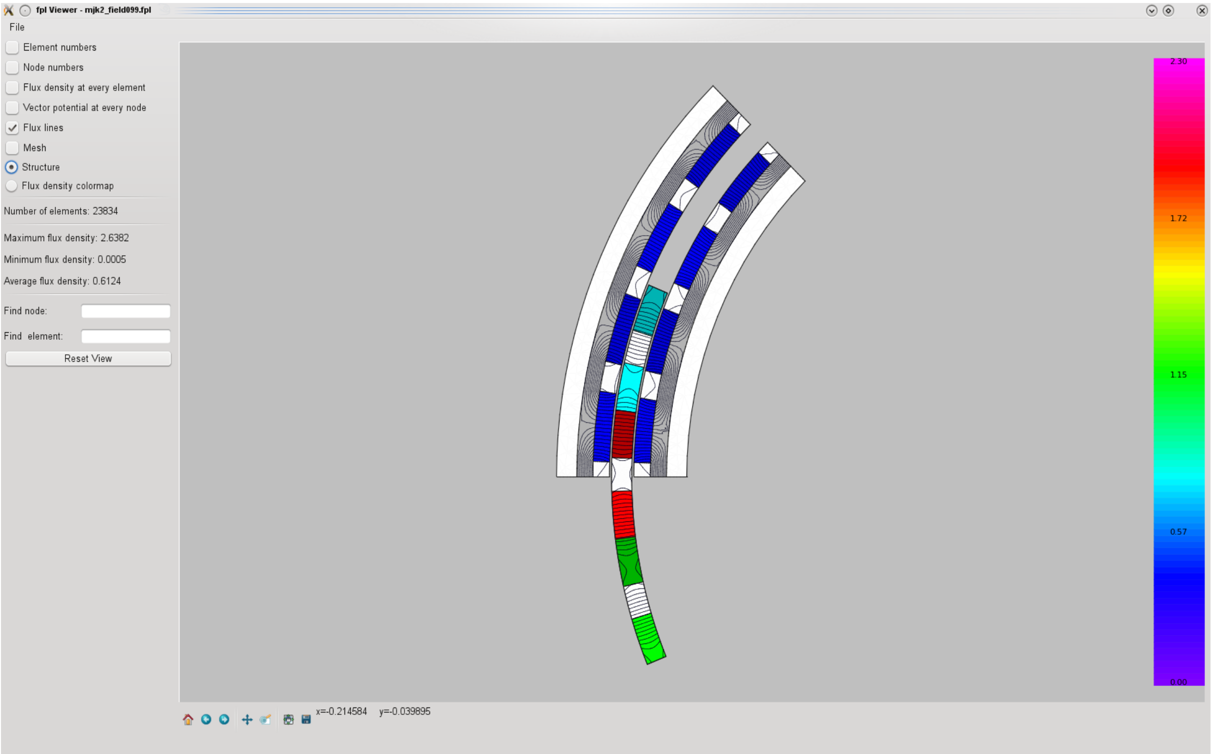


Figure F.2: Structure with movement.

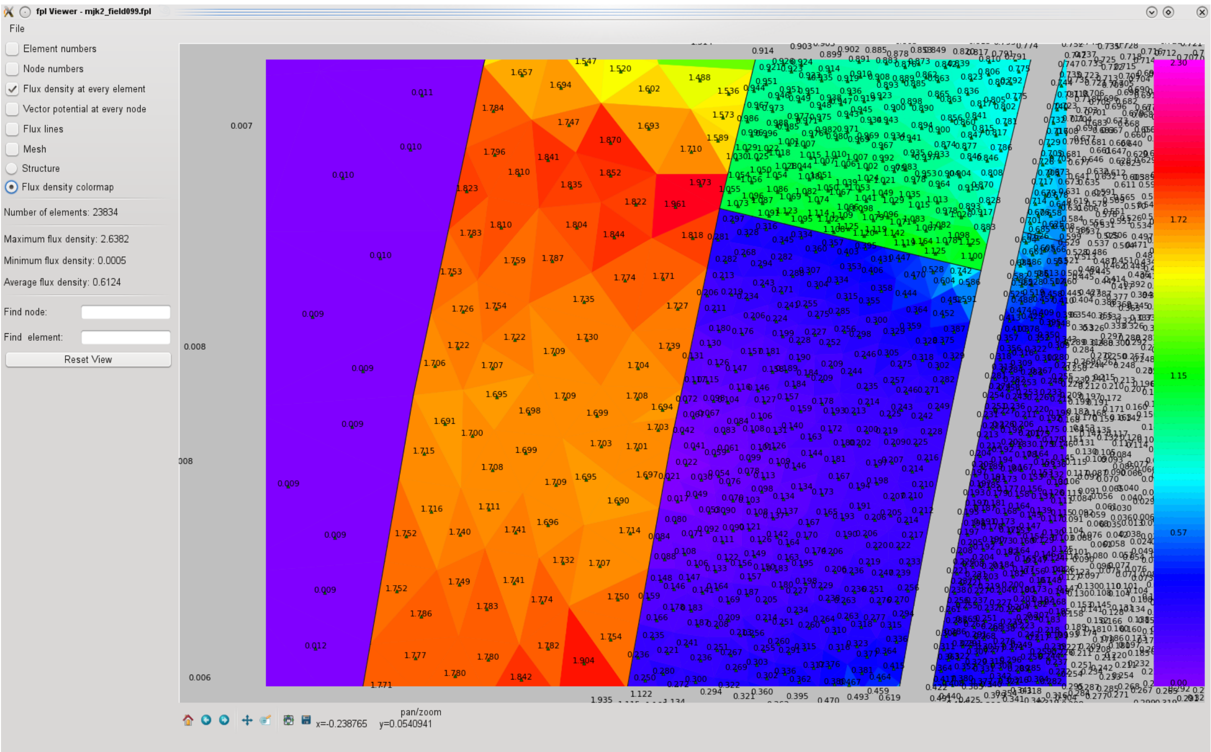


Figure F.3: Numerical values of the flux density at each element.